DESIGN PATTERNS IN GARBAGE COLLECTION

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science in Computer Science
in the
University of Canterbury
by

Stuart Andrew Yeates

University of Canterbury
1997

The world is filled with Knowledge;
it is almost empty of Understanding.

Louis Sullivan

Abstract

This thesis presents an examination of design patterns within the context of garbage collection. Initially, I review garbage collection and design patterns. Four garbage collectors are then examined and the design patterns found described. Both domain specific and generic patterns are described. The domain specific patterns are TriColour and RootSet, the generic patterns are Adaptor, Facade, Iterator and Proxy.

It is hoped that by, applying these patterns, systems designers have access to a less efficient, but simpler and more flexible way of implementing and reusing garbage collectors in programming languages.

The requirements analysis for a garbage collector for a real-time object-oriented microkernel is then performed, and a design prepared using the design patterns found in the other garbage collectors. The garbage collector is then implemented in Java using appropriate data structures. Due to timing difficulties in the runtime environment, timing was ruled out as a method of performance analysis. Algorithmic analysis is performed to evaluate the worst-case performance of the collector, which is found to be satisfactory in all but one method of the RootSet implementation. An approach to remedying this is suggested.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> We think it is likely that the widespread use of poor allocators incurs a loss of cache memory (and CPU cycles) upwards of a billion (10^9) U.S. dollars worldwide—a significant fraction of the world's memory and processor output may be squandered, at huge cost. (Wilson *et al.*, 1995)

This thesis draws on two areas of computer science, memory management and design patterns. Memory management presents a set of problems which have been studied for more than 30 years, and to which several solutions have been found, each useful in a particular domain. As the quote at the start of this chapter points out, however, many memory managers in end-user products don't use well-known, long-standing, 'good' solutions. Design patterns are a new field of computer science aimed at capturing the considered decisions of systems designers for later reuse or re-evaluation. Until now, to the best of my knowledge, design patterns have not been looked for in memory management, or in garbage collection, a specific sub-field of memory management.

1.1 Aim

The aim of this thesis is to research, design and implement a garbage collector for **OpenKernel** a real-time object-based micro-kernel (de Champlain, 1996b) using design patterns and written in Java. The garbage collector is to be specified and designed using object-oriented techniques and design patterns. The garbage collector is to perform in real-time, preferably hard real-time, and be de-coupled from the memory allocation and type subsystems.

1.2 Methods

The first step is to extract design patterns from existing garbage collectors. There exists a scarcity of object-oriented systems in the garbage collection field, even such high-profile object-oriented systems as the Oberon system resort to non-object-oriented assembly language for their garbage collection (Wirth & Gutknecht, 1990). As a result of this all of the garbage collectors examined for design patterns are non-object-oriented, requiring an object-oriented redesign before the design patterns can be captured.

The design patterns will be presented in 'Alexander' form.

Two garbage collector will be implemented in Java, but their performance proved difficult to evaluate. The Java virtual machine and compiler which I am working with displays unusual characteristics, for example, static method calls were slower than non-static method calls, but only on some CPUs. As a result of these problems, algorithmic worst-case analysis is used to evaluate the performance of the collector rather than measured speed.

1.3 Results

During my investigation I discovered six design patterns. Four of the patterns discovered, Facade, Adaptor, Iterator and Proxy, were generic patterns used in domain specific ways. Two of the patterns discovered, TriColour and RootSet, are domain specific to garbage collection, and unlikely to be found elsewhere. Of these patterns probably the most interesting is TriColour, representing the proof-of-correctness in incremental garbage collectors, which has been described in the literature, and the documentation of garbage collectors, for 30 years, but has, hitherto, not been described in object-oriented terms, nor as a design pattern.

Two garbage collectors were implemented using the same group of patterns, an incremental mark-and-sweep and a simple generational collector. Of these the generational collector was the more complex, both in terms of lines of code and in terms of algorithmic complexity. Both collectors gave real-time performance in all except one method call, in the RootSet implementation. A scheme to rectify this, at the cost of considerably closer coupling between the execution stack and the collector, is proposed.

1.4 Thesis Outline

In chapter 2 I examine garbage collection theory and algorithms in some detail, reviewing previous work. In chapter 3 I explore garbage collection as it applies to different languages and environments. Chapter 4 presents an introduction to object-orientation and design patterns, with many examples. Chapter 5 presents object-oriented analyses of four independently designed and constructed garbage collectors. Chapter 6 presents the design patterns found in these collectors, Adapter, Facade, Iterator, Proxy, RootSet and TriColour within the context of garbage collection. Chapter 7 presents my own design for a fully object-oriented collector using these patterns, and an analysis of it's worst case performance.

Chapter A is a glossary of common terms in garbage collection. Chapter B contains discussion of resistance to interference in iterators, a serendipitous sidetrack I travelled while researching for this thesis.

1.5 The scope of this thesis

Modern object-oriented memory system developments have branched in two directions, those which limit themselves to a single "main" memory and those which consider memory in a more general sense, either spread across several computers (a distributed memory) or across several media (for example persistent object stores which save memory to disk). The later has seen developments in networking, persistent heaps, object databases and distributed shared memory systems which require paging, caching and fault tolerance to be considered. The former make the simplifying assumption of a single non-networked processor executing with no hardware faults in a single flat address space; it is with these that this thesis deals.

The invention of new algorithms is also beyond the scope of this thesis. Existing implementations and algorithms are examined and an appropriate algorithm selected.

The view of design patterns taken in this thesis is a narrow one, approximately corresponding to that put forth in (Gamma *et al.*, 1995) and (Buschmann *et al.*, 1996), which focus on the interactions of objects within the software system under consideration.

1.6 A note on originality

The bibliography for this thesis is based in part on a seven page survey performed by Marcel van Mierlo, during the summer of 1995-96 (van Mierlo, 1996).

Java source code in chapters 2, 3 and 4 is original. Source code examples are taken from the Boehm collector in chapter 6, but all other source code in that chapter is my own work. Source code in chapter C is drawn from a system designed in conjunction my supervisor, Michel de Champlain, where source code is his work, or a combination of our efforts, this is noted.

The central original contribution of this thesis is the design patterns found in chapter 6.

Chapter 2

Garbage Collection

> The fundamental question about memory management can be stated in this way: If function f() passes or returns a pointer to an object to g(), who is responsible for the objects destruction? [...] ideally "the system" (Stroustrup, 1986)

2.1 Traditional Memory Management

Traditional memory management on modern computers divides the memory into three main areas: global data, the stack, and the heap, as shown in figure 2.1. The *global data* area is a collection global variables, it is of constant size and globally visible throughout the program's execution. The *stack* contains a series of frames, each representing a function scope. Both of these areas contain references, called *roots*, into the *heap* which holds chunks[1]. Chunks are used for application dependent purposes and may contain arbitrary pointers

Chunks on the heap are generally allocated and de-allocated explicitly, an application must be able to detect when a chunk is no longer needed so that it may be de-allocated and the memory reused. The structure that manages de-allocated memory waiting for re-allocation is called a *free list*, though often implemented as an array of lists or a tree of some variety (Nilsen & Gao, 1995). Stack frames are de-allocated when the procedure associated with them terminates.

[1] some authors use the term 'object' to refer to blocks of memory allocated and used in an application dependent manner. The term 'chunk,' however, emphasises the difference between these and object-oriented 'objects' which also have methods and a type associated with them.

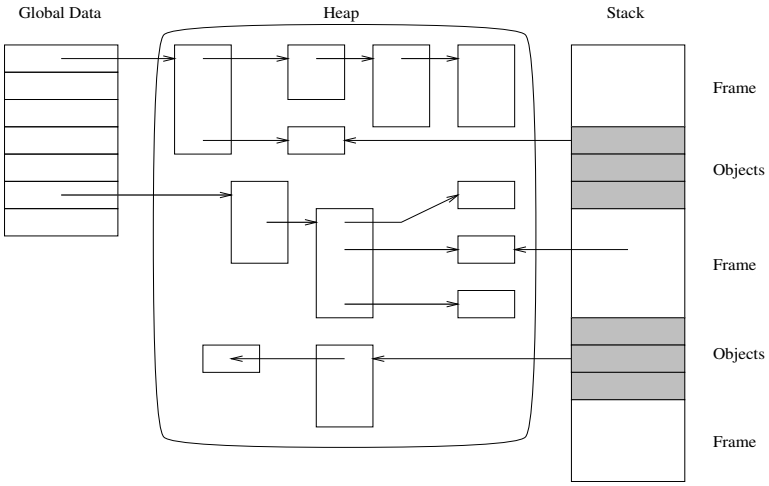


Figure 2.1: The traditional memory management model

2.1.1 Problems

This form of memory management has several long standing problems, which have been widely discussed elsewhere (George O. Collins, 1961; Nilsen, 1994; Guggilla, 1994). These problems, fall into three main categories:

1. Incorrect program decisions as to whether a chunk is still in use lead either to memory leaks (if unneeded chunks are kept) or to dangling pointers (if needed chunks are de-allocated). Both of these can lead to catastrophic results in non-trivial programs.

2. The structure and types of chunks on the heap are highly dependent on the domain of the application and the language in which it is implemented. In general the types of chunks are not determinable by examination of the chunks themselves.

3. Allocation and de-allocation using traditional algorithms such as first-fit, binary search tree or partitioning by size can be very expensive in terms of processor time and memory accesses (Nilsen & Gao, 1995).

Additionally, since it forces routines which operate on a chunk to be aware of other routines operating on the same chunk in order to make de-allocation decisions, traditional memory management impedes truly modular programming (Guggilla, 1994).

2.2 Garbage Collection

Garbage collection offers an alternative to traditional memory management. Instead of the application task or *mutator* (Dijkstra *et al.*, 1978) explicitly de-allocating chunks when they are no longer needed, the garbage *collector* uses program state information to determine whether any valid sequence of mutator actions could reference a chunk. If no valid sequence of program actions can reference a chunk, then it is unnecessary in the on-going computation, the chunk is *garbage* and may be reclaimed by the collector. Conversely, all chunks which can be referenced by such a sequence of actions are *live*.

Garbage chunks may be isolated, containing no references to other garbage, or they may form lists, trees or other structures of potential arbitrary complexity with other chunks, both free and live.

Garbage collection also implications for the operation of execution stack. Studies such as (Appel & Shao, 1994) indicate that the cost of allocating stack frames and associated chunks on a garbage collected heap can be comparable the cost of traditional stack allocation. This technique is especially useful for languages with closures, such as Scheme, in which frames can have a lifetime independent in the lifetimes of their callers and callees (those immediately 'above' and 'below' them on the stack).

Modern surveys of the garbage collection field can be found in (Corporaal & Veldman, 1991; Wilson, 1992; Nilsen, 1994), and (Sanaran, 1994; Wilson *et al.*, 1995) present an extensive bibliography. (Jones & Lins, 1996) presents a through exploration of the field including code samples.

2.2.1 Garbage collection as a graph traversal problem

Objects which can be referenced by a future sequence of valid mutator actions are said to be reachable. Those which will be referenced in the future by the mutator are said to be live, that is, necessary to the ongoing computation.

A garbage collector can be thought of as performing a liveness analysis on heap objects, locating those chunks which cannot be referenced either directly from a *root set* (the stack and global variables), or from a root set via other chunks on the heap. Figure 2.2 shows that all live chunks may be reached from a root set and that no garbage chunk may be reached. Transforming garbage collection into a graph traversal problem enables traditional graph traversal algorithms such as depth- and breadth-first searching to be utilised when *tracing* the heap to decide whether a chunk is garbage. (Wilson *et al.*, 1991) presents a discussion of the relative merits of various tracing algorithms.

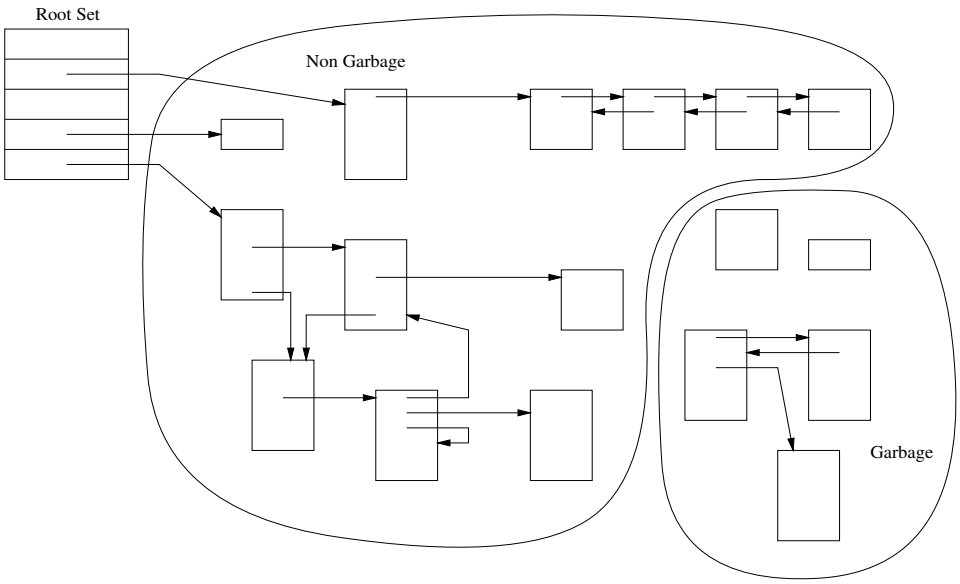


Figure 2.2: Garbage collection as graph traversal

Such liveness analysis is, however, inherently undecidable, because the Church-Turing hypothesis (Epstein & Carnielli, 1989) prevents determination of the liveness of objects in some situations.

Consider, for example, the following Java class:

```
import java.lang.Object;

/** A class to illustrate the undecidability of reachability analysis */
class Undecidable {

  /** A method whose completion is undecidable */
  void anUndecidableMethod(){
    // ...
  }

  /** A method which illustrate undecidable reachability. */
  void anotherMethod(){

    // the object whose reachability is undecidable
    Object anObject = new Object();

    // the method call during which anObject reachability is undecidable
    anUndecidableMethod();
```

10

```
    // an access of anObject                                    20
    anObject.getClass();

    // another call method call
    anUndecidableMethod();
  }
}
```

---

anUndecidableMethod() is a method whose completion is an undecidable problem, in the Church-Turing hypothesis sense. aMethod() is another method, which creates a new Object and then calls anUndecidableMethod() in line 18. Object is only accessed in line 20, which will never be executed if anUndecidableMethod() never returns. Since anUndecidableMethod()'s return is undecidable, so is the execution of line 20, and the access of anObject(). If anUndecidableMethod() doesn't return, then anObject is not live, and may be collected, immediately after creation. If anUndecidableMethod() does return, anObject is live, and may not be collected, until the completion of it's access on line 21.

In practice, all known garbage collectors assume that all method calls return. This is a conservative assumption, it may lead to the retention of reachable but not live objects, but never to the freeing of reachable or live objects.

Most garbage collectors also assume that anObject is live during the call on line 24, even though no code between line 21 and 25 accesses it. This is also a conservative assumption, but it greatly increases the usefulness of debuggers, as variables hold their value until they pass out of scope, rather than having undefined value between the last executed reference to them and when they pass out of scope.

These two assumptions convert liveness into scope, a well understood notion fundamental to modern programming language design. Some work has been done on the effect of these two assumptions and it appears that in some cases, considerable memory is lost to them. For example, consider the following Java class, which accepts a number of command-line arguments, reads them into variables and then performs some processing based on their values:

```
/** A class to illustrate retention of excessive memory */
class AClass {

  /* A number of state variables whose value is determined by the command line arguments to the program */
  static boolean _stateInformationA;
  static boolean _stateInformationB;
```

```
  //...

  /** A method to read the command line arguments into the state variables. */
  static private void handleCommandLineArgs(String argv[]){                    10
    // ...
  }
  /** A method which actually does the programs work */
  static private void doStuff(){
    //...
  }
  /** The entry point into the program */
  static public void main(String argv[]){

    // handle the command line arguments                                       20
    handleCommandLineArgs(argv);

    while (true){
      doStuff();
    }
  }
}
```

The program spends the vast majority of it's time in the while loop in lines 27-29, by which time the command-line arguments, stored in **argv**, are known to be redundant (unnecessary for ongoing computation). While retention of such objects are necessary when a debugger is running (or in a system, such as the Smalltalk runtime environment, where the system debugger may be invoked at any time), in the general case, they may be reclaimed.

2.2.2 Tri-colour graph colouring

Tri-colour graph colouring, an abstraction first described in (Dijkstra *et al.*, 1978), is useful in developing proofs of correctness for garbage collectors, especially those which are incremental or concurrent (Baker, 1992; Wilson, 1992; Virding, 1995). The algorithm can be stated as follows:

1. Initially, all roots (chunks pointed to from outside the heap) are coloured grey, and all other heap chunks are coloured white.

2. Examine a grey chunk,

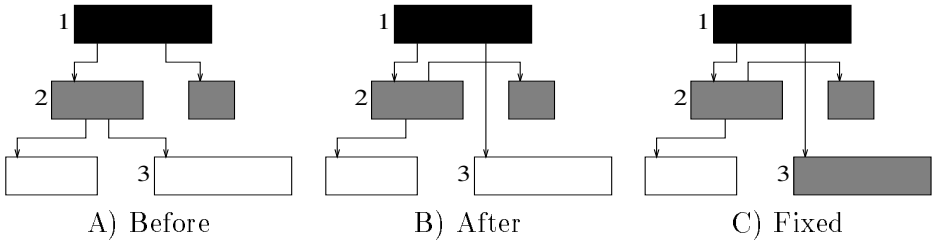


Figure 2.3: A violation of the colouring invariant, (A) before and (B) after the mutator has exchanged two pointers. C shows the state after operation of a write-barrier.

(a) Colour it black.

(b) If a pointer leads from it to a white chunk, colour the white chunk grey.

3. Repeat step 2 until no grey chunks remain.

4. After colouring, all live chunks are black, all remaining white chunks are garbage, and may be reclaimed. There are no grey chunks.

The algorithm may be visualised as a wave of black moving through the initially white heap, with a grey chunk buffering the black from the white at all times (Wilson, 1992).

During garbage collection, no pointer leads from a black chunk to a white chunk. The importance of this invariant is that the collector must be able to assume that black chunks are "finished with" and that it need only traverse grey chunks and move the wavefront forward. If a mutator, operating concurrently with the collector, somehow creates a pointer from a black chunk to a white one, the collector must ensure, in some manner, that the collector's bookkeeping is brought up-to-date.

Figure 2.3 shows an example of mutator action during garbage collection. The action consists of changing of a pointer in chunk 1 which before pointed from chunk 2 to point to chunk 3. While before the mutator action (A) the colouring invariant is preserved, afterwards (B) the pointer from chunk 1 to chunk 3 breaches it. If left in this state, chunk 3 would be incorrectly reclaimed. (C) shows the state after operation of a write-barrier, which has coloured chunk 3 grey, to ensure that it is traced.

It should be noted that the tri-colour graph colouring presented here is a quite different tri-colour graph colouring to that presented in (Washabaugh & Kafura, 1990), which uses white to colour roots, grey for non-root live chunks, white being used in the same sense as I use black.

(Dijkstra *et al.*, 1978) showed that all incremental garbage collectors must provide a method of synchronising the collector and mutator to preserve the colouring invariant.

There are two main methods of preserving this invariant: read-barriers and write-barriers. A read-barrier ensures that no white chunks are seen by the mutator by colouring grey any white chunks about to be read; this can lead to a flurry of writes immediately after the start of garbage collection (Dijkstra *et al.*, 1978). A write-barrier ensures that after a black chunk is written to, it is either coloured grey or traced (Boehm *et al.*, 1991). Because writes are generally less frequent than reads, most implementations use a write-barrier. (Boehm *et al.*, 1991) uses the memory protection facilities of modern virtual memory hardware to implement a write-barrier, making it very efficient, while (Magnusson & Henriksson, 1995a) gets the write-barrier down to a mere 12 machine instructions using in-lined code executed at every write to a chunk.

2.3 Conservative assumptions in Garbage Collection

Several simplifying conservative assumptions can be made in garbage collection:

1. All functions are assumed to return. This assumption is universal to all garbage collectors, without it, reachability analysis is equivalent to the halting problem (Epstein & Carnielli, 1989).

2. Depending on the nature of the compiler in use, objects reachable from variables which are current but which may be shown by static analysis to be unnecessary may be assumed to be reachable. This is an area in which compilers which are aware that a garbage collector may be in use can aid them even without direct communication.

   These first two assumptions are discussed in (Boehm & Weiser, 1988) and in Section 2.2.1.

3. Because type information of rapidly changing registers can only be maintained in expensive, non-portable ways, some implementations make the conservative assumption that all registers are pointers. This may lead to memory retention, but due to the rapid changes which can be expected in registers, it is unlikely that the values will be identical sequential collections, causing memory to be retained for at most one collection, leading to an upper bound on retained memory.

4. Execution stacks have similar problems to registers, but change less frequently, making them both cheaper to maintain type information for and increasing the memory retention problems (Wentworth, 1990).

5. In languages with little type safety, such as C/C++, conservative assumptions must be made about every byte of every object on the heap.

These last three assumptions are, effectively, successive generalisations of a single assumption, in which the implementation trades off lack of type information for increased memory retention. The term 'conservative collector' is applied to those collectors which make some or all of these three assumptions, the term 'accurate collector' is applied to those which make none of them. The effectiveness of this tradeoff is dependent on four factors:

1. Pointer size. A short pointer (16 or 32 bits) leads to a relatively high density of objects in the address space, leading to a high probability of a random bit-pattern being mistaken for a a valid pointer into the heap.

2. Volatility of the data. If an object if allocated at an address apparently pointed to be static data, it will be 'pinned' for the lifetime of the program, while changeable data will eventually point elsewhere, allowing such a pinned object to be freed. The Boehm collector (see chapter 5.3) has a novel method for overcoming this called blacklisting.

3. Type maintenance cost. The cost of maintaining type information for the given area of memory varies between registers (very high cost of type maintenance) and the heap (relatively low cost of type maintenance). In a non-type-safe language the cost of maintenance is extremely high, relying on global static analysis and other techniques which are rarely seen in practise.

4. The interconnectedness of chunks. Large, multi-cyclic data structures are more likely to be retained, and retained for longer, than single elements objects with no pointers to them from other garbage.

The selection of which of these assumptions an implementation will make appears to be largely dependent on the type-safety of the target language. Implementations for type-safe languages (such as Java and Scheme) rarely make assumptions 3, 4 or 5, whereas implementations for non-type-safe languages (such as C and C++) are forced to.

2.4 Garbage Collection Algorithms

2.4.1 Reference Counting

One of the oldest garbage collection algorithms, reference counting (Deutsch & Bobrow, 1976) requires that a count be kept of the number of inward references to each chunk, either in the chunk (see figure 2.4), or in a separate table. When the last reference to a chunk is removed, it can no longer be reached by the mutator and is garbage. As shown in figure 2.5, reference counting algorithms are unable to collect circular structures, since all chunks in these structures in the circle still have inward references, but are unreachable by the mutator. While a number of extensions have been proposed to enable collection of circular structures, most are essentially hybrids of reference counting and the other algorithms presented here (Guggilla, 1994).

The UNIX inode is a classic example of the use of reference counting is a specialist domain, which is also used by such modern programs as Abode's Photoshop for memory management (Jones & Lins, 1996). In these cases, there is a tight coupling of the application and memory management, and a need for a prompt return of the potentially scarce resource.

2.4.1.1 Advantages and Disadvantages

Reference counting is unable to defragment memory or reclaim circular structures. Chunks are accessed as references to them are created or deleted, no sweeping of chunks is necessary. No run-time type information is required. Very good performance in a paged environment due to non-sweeping action.

2.4.2 Mark-and-Sweep

The mark-and-sweep collection cycle has two phases. In the marking phase, all reachable chunks are marked as reachable. During the collection phase, unreachable chunks are reclaimed (McCarthy, 1960).

Since it's first implementation, a wide range of improvements and optimisations for mark-and-sweep have been proposed and implemented, including division of the according to chunk size for efficiency (Heeb & Pfister, 1990?) and language independent implementations (Ferreira, 1991). Mark-and-Sweep is also amenable to various forms of concurrency (Dijkstra *et al.*, 1978; Baker, 1978; Washabaugh & Kafura, 1990; Kuechlin & Nevin, 1991; Boehm *et al.*, 1991; Tanaka *et al.*, 1994). Most implementations use a timestamp to mark chunks mutated during the relatively long marking phase, allowing both the mutator

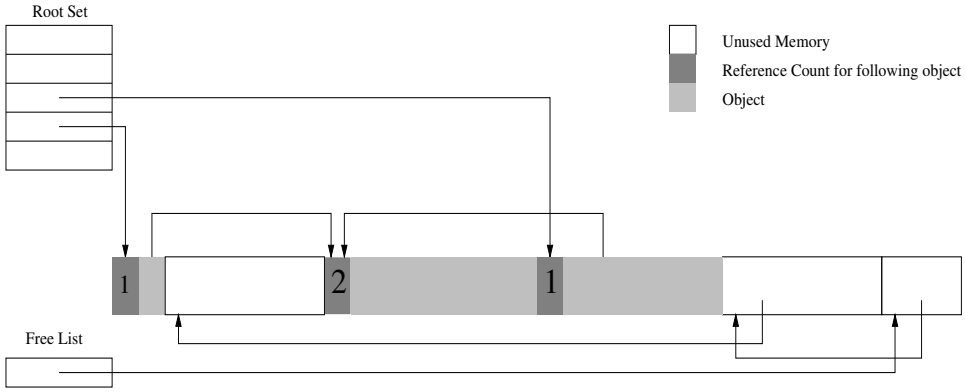


Figure 2.4: The heap organisation using the reference counting algorithm

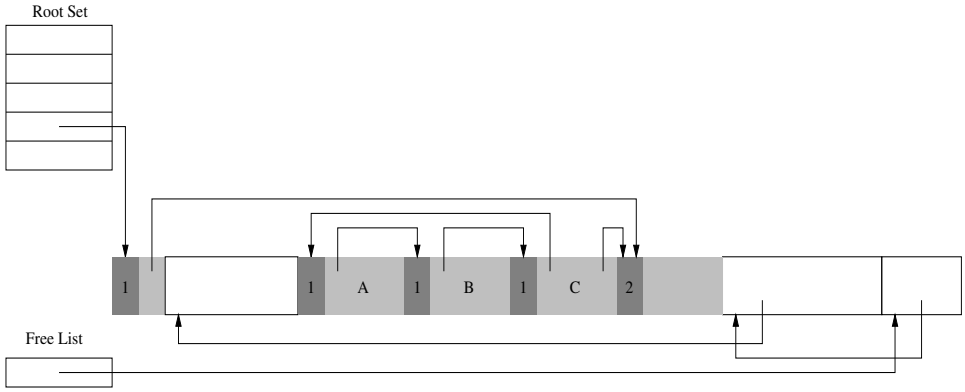


Figure 2.5: The heap using reference counting with unreachable (never reclaimed) chunks

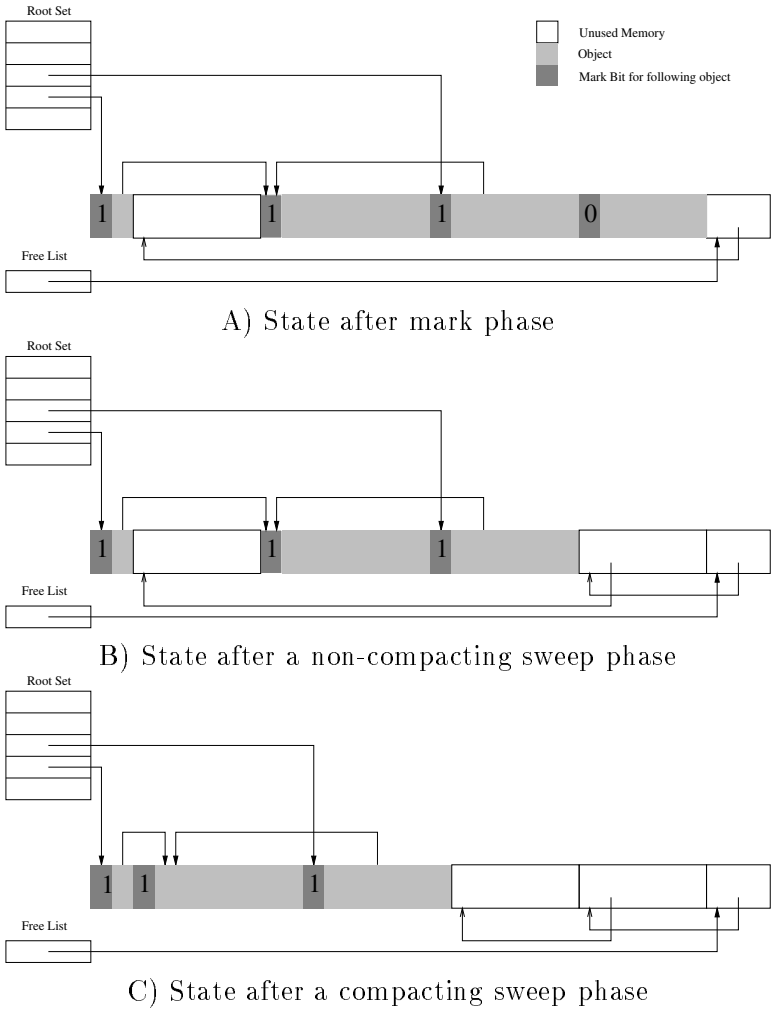


Figure 2.6: The heap using Mark-and-Sweep (A and B) or using Mark-and-Compact (A and C)

and the collector free access to the heap. Generally the mutator is stopped or fine grained synchronisation is used for the generally shorter collection phase. Modern systems use such features as virtual memory hardware to detect chunks which have been written to since the start of the cycle (Boehm *et al.*, 1991).

2.4.2.1 Advantages and Disadvantages

Mark-and-Sweep is unable to defragment memory, but reclaims circular structures. All chunks are swept twice every cycle, once in the marking phase, and once in the sweeping phase.

2.4.3 Mark-and-Compact

A variant of the mark-and-sweep, called mark-and-compact, overcomes the problem of memory fragmentation by compacting reachable chunks in the collector phase, to remove the space between them (Steele, 1975).

Figure 2.6 shows the process of a mark-and-sweep garbage collection. Figure 2.6 a shows the heap after a mark phase, (B) after the sweep phase, and (C) after the same heap after the operation of a mark-and-compact collector.

2.4.3.1 Advantages and Disadvantages

Mark-and-Compact is able to defragment memory and to reclaim circular structures. It's incremental variants require mutator checks on chunk reference during the collection cycle. As with mark-and-sweep, all chunks are swept twice (Washabaugh & Kafura, 1990; Tanaka *et al.*, 1994).

2.4.4 Semi-Space

Semi-Space garbage collection algorithms divide the heap into two contiguous areas called the from-space and to-space. For each garbage collection cycle all reachable chunks are traced and copied from the *from-space* to the *to-space* (Fenichel & Yochelson, 1969). The garbage collection begins with a 'flip,' which renames the from-space to the to-space and *vice versa*. Each time a new chunk is allocated, a fixed number of chunks are copied, usually breadth first, from the from-space to the to-space. As each chunk is copied, all pointers within it are adjusted to point to new chunks in the to-space. If the chunk being pointed to has not yet been copied, space is reserved for it in the to-space. The cycle is complete when all chunks have been copied. New chunks are allocated in the to-space, and when the to-space is full, another cycle is started (Guggilla, 1994; Corporaal, 1991a).

If the semi-space collector is incremental (copying only a portion of chunks before returning control to the mutator), the mutator must be able to determine which copy of the chunk to use, that in the from-space or that in the to-space. This is determined by the value of the forwarding pointer field in the chunk header (see figure 2.7), and may require one level of indirection every time a chunk is referenced by the application during garbage collection, but only until all chunks have been copied.

Figure 2.4.4, (A), (B), (C) and (D) show the operation of an incremental semi-space copying algorithm, the most common form of the semi-space copying algorithm. The

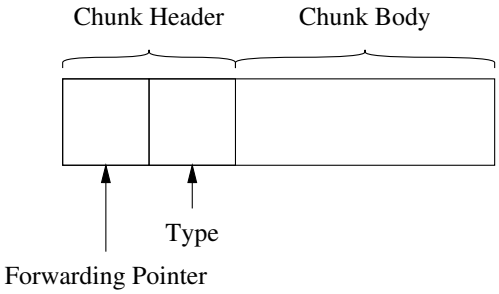
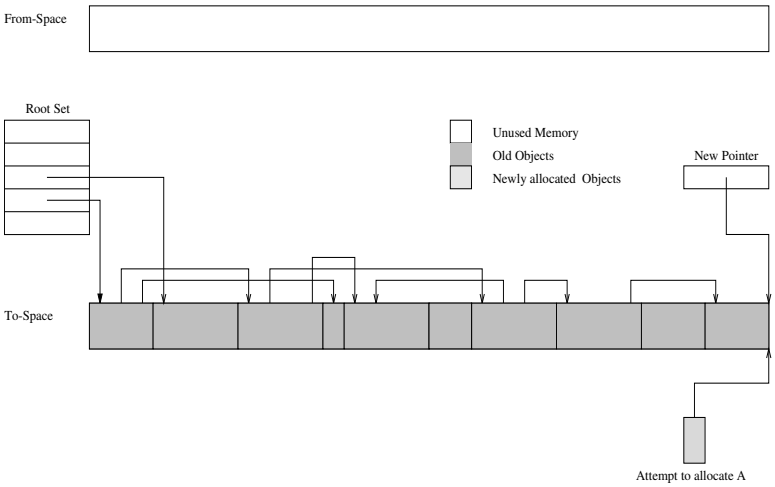
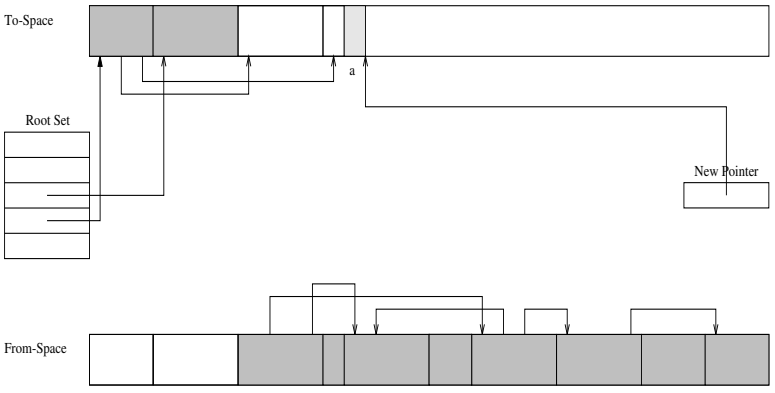


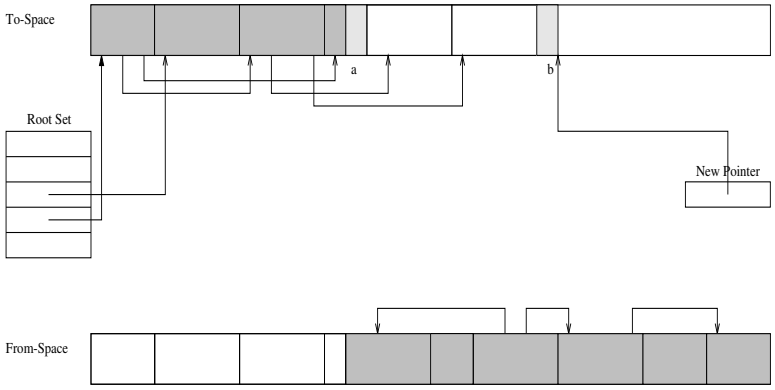
Figure 2.7: The structure of the header used in semi-space copying



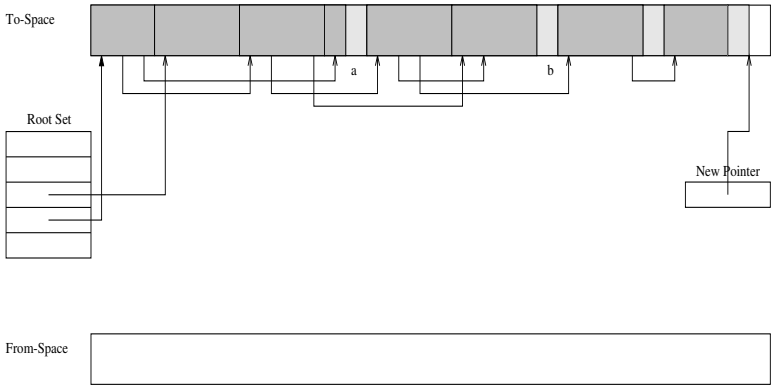
A) The heap at the start of the cycle



B) The heap after copying first two chunks



C) The heap after copying first four chunks



D) The heap at completion of cycle

Figure 2.8: The heap using semi-space copying

algorithm uses a chunk header (figure 2.7), which contains the the address of the chunk in the other semi-space, and the chunk type, from which it can derive the chunk size.

(A) shows the state of a semi-space collector immediately before the start of a new garbage collection cycle. The cycle is triggered when the new pointer, the address for the allocation of new chunks, reaches the end of the semi-space. The first step in the garbage collection cycle is a *flip*, the from-space and the to-space are (logically) exchanged, so that the to-space in the last cycle is the from-space in this cycle while the from-space in the last cycle is cleared to become the to-space in this cycle.

The next step is the copying from the from-space to the to-space of chunks pointed to by the root set, as shown in (B). As chunks are copied, their forwarding pointer fields in the to-space are set to point to themselves, and all pointers within the chunk are adjusted to point to new locations of chunks. If the chunk pointed to has yet to be copied, space is reserved for it by incrementing the new pointer by the chunk size, and the address of the chunk in the from-space is inserted in the forwarding pointer field in first byte of the reserved space. Similarly, the address of the reserved space is placed in the forwarding pointer field of the chunk in from-space. When a chunk which had has space reserved for it is copied, the forwarding pointer in the to-space is set to point to itself. The new chunk (a) which triggered the garbage collection cycle is then allocated, and control returned to the application.

The next chunk allocation (a) causes another two chunks to be copied, as shown in (C). (D) shows the state of the collector when all the chunks have been copied.

The time taken to allocate a new chunk is proportional to the number of chunks copied on each allocation. Any pointer to a chunk used by the main program during the collection must be checked to ensure that once the chunk has been copied to the to-space, the pointer points to the to-space. This overhead is constant for each reference, but is not incurred once all chunks have been copied. Thus the more chunks copied on each allocation, the higher the allocation overhead but the lower the reference overhead (Guggilla, 1994; Baker, 1978).

Because the algorithm deals exclusively with reachable chunks, in the absence of destructor methods the speed of the algorithm is proportional to the number and size of non-garbage chunks, and is unaffected by the size of the heap (Guggilla, 1994).

2.4.4.1 Advantages and Disadvantages

Semi-Space collection is able to defragment memory, and reclaims circular structures. It's incremental variants require mutator checks on chunk references during the collection

cycle, and it requires a run-time type system. All live chunks are swept at least once, most twice, during each collection.

2.4.5 Generational

Generational algorithms use spaces to hold objects based on their age. The youngest chunks, those in the new generation, are managed in a similar fashion to semi-space garbage collection. After surviving a fixed number of garbage collections in the new generation, chunks are 'promoted' to an older generation. A list of intergenerational references is held, enabling the older generation(s) to be managed separately. Because the younger generations are smaller, and a higher proportion of objects in them die each cycle, less work total is required.

Treatment of the older generation differs markedly among implementations. (Ungar, 1984) describes an implementation in two generations (old and new), in which no garbage collection in the old generation occurs at run-time. By setting a tough promotion criteria and assuming that all chunks that meet the criteria will be live indefinitely, the author did away with collection of the older generation entirely. He reports as little as 0.2 % of chunks were promoted incorrectly. (Seligmann & Grarup, 1995) also has two generations and uses the 'train' ordering algorithm, an elaboration of reference counting. This algorithm groups chunks into ordered, disjoint sets — 'trains' — in such a way that circular structures are grouped into the same train. When the last reference to a train is deleted, the train is reclaimed. This algorithm, while very useful has several drawbacks, including poor handling of very *popular chunks*, to which there are a large number of references, and a low speed of garbage collection in the face of misguided promotion criteria.

Figure 2.9 shows the heap layout for a generational algorithm.

2.4.5.1 Advantages and Disadvantages

Generational collection is able to defragment memory, and reclaims circular structures. It's incremental variants require mutator checks on chunk reference during the collection cycle, and it requires a run-time type system. It sweeps all chunks in the youngest generation at least once, most of them twice, during each collection.

2.4.6 Treadmill

In (Baker, 1992) Baker describes the treadmill. At the cost of not defragmenting memory, the treadmill converts the semi-space (or generational) copying algorithm to a non-copying one, saving the costs of copying areas of memory. Instead of defining the

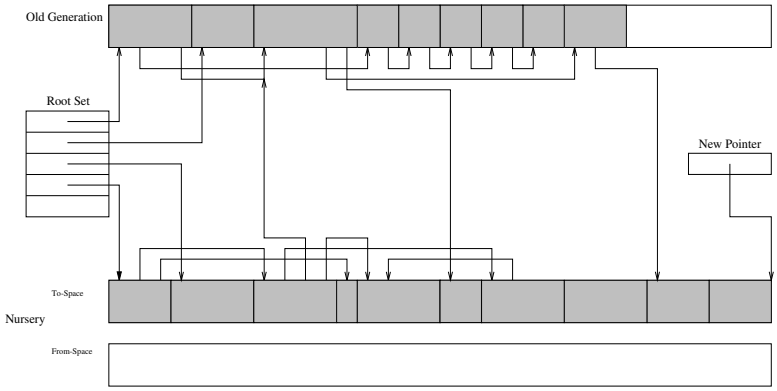


Figure 2.9: The heap of the generational garbage collection algorithm

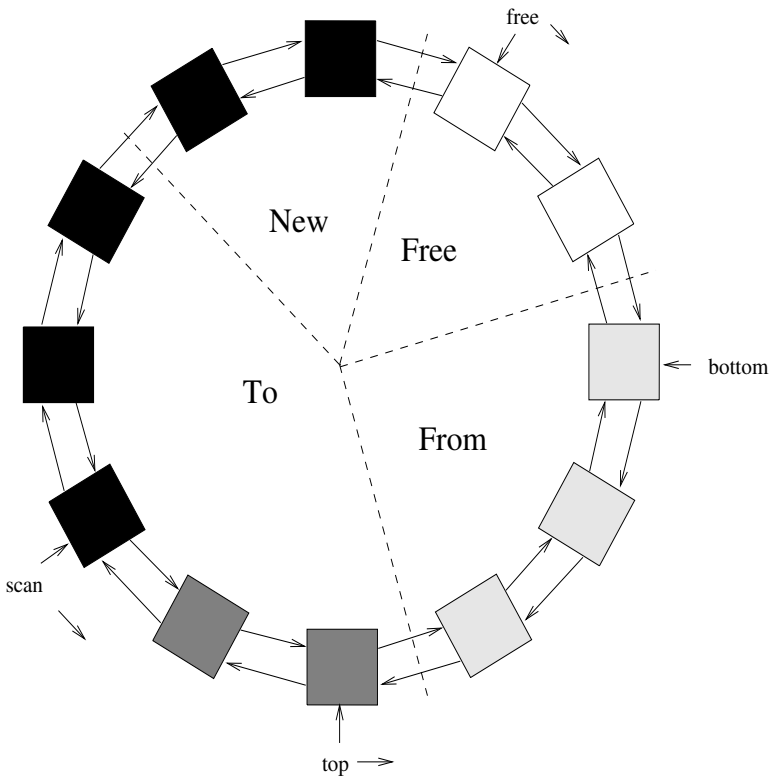


Figure 2.10: The structure of the Treadmill doubly-linked list

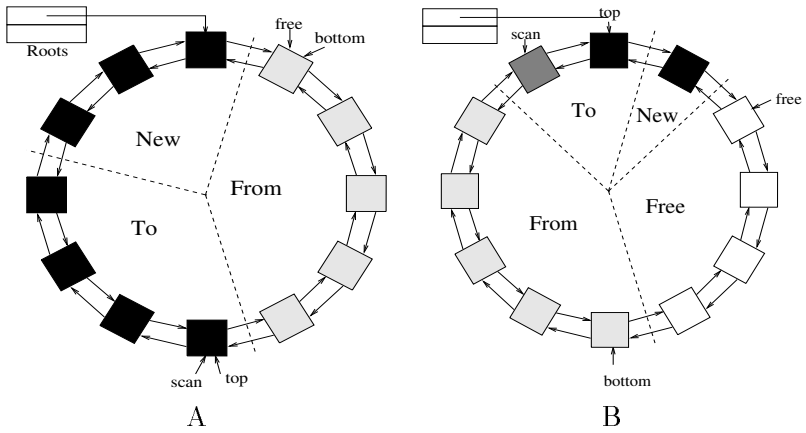


Figure 2.11: The Treadmill list immediately (A) before and (b) after a flip

semi-spaces as contiguous ranges of memory, the treadmill optimisation uses a circular doubly-linked list to create logical spaces. The semi-spaces are defined as contiguous ranges of the list.

For simplicity's sake, I shall examine the semi-space version of the treadmill, using chunks of uniform size.

Figure 2.10 shows the structure of the doubly-linked list. An extra colour is needed (in addition those defined in section 2.2.2) because the list manages a free-list as well as the memory in use, hence ecru (French for "off white") is added to the palette. White chunks (□) are those on the free-list, ecru chunks (□) are those in the from-space. Grey (■) and black (■) chunks have the same meaning as in section 2.2.2.

The pointers free, top and scan in figure 2.10 have a directional arrow, indicating their direction of movement during the garbage collection cycle. bottom does not move except during the flip.

When a chunk is allocated, a quantum of garbage collection is performed, free moved to the next item and the chunk previously pointed to by free is returned. In this way the chunk is moved from the free area to the new area, which holds chunks allocated during the current garbage collection cycle.

Figure 2.11 shows the state of the list immediately before and after a flip. The from-space (ecru) of the last cycle becomes the free-list (white) of this cycle. The to-space (black) of the last phase becomes the from-space (ecru) of this phase. While it may appear that recolouring requires scanning every coloured chunk, this may be implemented by changing the meaning of bit masks representing colour. The roots are traced, the new

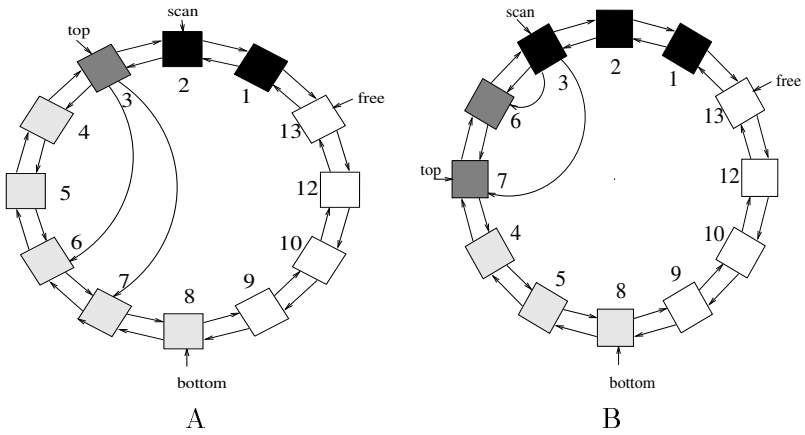


Figure 2.12: The treadmill immediately (A) before and (b) after chunk 3 has been traced and marked

chunk allocated and control returned to the mutator.

Figure 2.12 shows the state of the list immediately before and after a quantum of garbage collection. The chunk in front of top (chunk 3) is traced and marked. First ecru chunks pointed to by 3's pointers (6 and 7) are removed from their position in the list and inserted back in after top. When scan points to the same chunk as top, the garbage collection cycle is complete.

The treadmill does not affect the need for the mutator to synchronise with the collector, as discussed in section 2.2.2.

2.4.6.1 Advantages and Disadvantages

The treadmill is unable to defragment memory, and allows many operations to be performed in constant time. In other respects, however, it can be implemented to be identical to either the semi-space or generational collectors.

2.4.7 Summary

Table 2.1 summarises the key attributes of each of the garbage collection algorithms discussed. Additionally, basic performance information of a typical implementation and a list of relevant papers is given. 'Memory Accessed' is important, because all collectors which sweep, or examine, large numbers of chunks pose problems with modern virtual memory systems. 'Program Work' is the number of references to heap chunks created or deleted, commonly used in the literature as a measure of the amount of work performed by an

application (Wilson, 1992).

2.5 The Baker algorithm

Baker (Baker, 1978) proposed that for each allocation, a certain amount of garbage collection should be performed, typically if n bytes are allocated, $2n$ bytes are copied to the to-space. In this way, an incremental semi-space or generational collector could be implemented which could both guarantee that not only would the garbage collection keep up with memory allocation, but both allocation and collection would be 'real-time' — their execution time would be bounded by a small upper bound.

There is, as one would expect, a cost. Because not all chunks are copied at once, the mutator must check for every chunk it references which semi-space it resides in. This overhead, which amounts to an indirection, is only incurred while a collection is underway, also has its execution time would be bounded by a small upper bound.

The number of chunks copied each allocation is a tunable parameter. The more chunks copied, the sooner the collection is over and fewer times the indirection overhead occurs. Copying more chunks also increases that upper bound on allocation execution times.

The key property of the Baker algorithm is that it can be used to guarantee that garbage collection will be finished before memory is exhausted while keeping all operations tightly bounded.

2.6 Preventive Garbage Collection

Preventive garbage collection, is the technique of performing garbage collection at a point in time at which the application is using minimal memory, in terms of both stack and heap (Kuechlin & Nevin, 1991). Because memory in use is at a minimum, the time taken to perform the garbage collection is minimised.

The extreme case of this is a thread or process acting as a server; it accepts a request and during processing it allocates memory to hold intermediate results. When the result is returned to the client, the entire heap may be reclaimed, as the intermediate results are no longer of use. A garbage collection at this point would reclaim the entire heap (Kuo & Kuo, 1993).

In less extreme cases, the application may, in some domain-specific way, know that it has reached a state in which it is using minimal memory. For example a text editor after closing a file, or a scheme interpreter after the completion of a program. A garbage

Scheme	Traditional Memory Management	Reference Counting	Mark-and-Sweep	Mark-and-Compact	Semi-Space Copying	Generational	Treadmill
Defragments memory?	No	No	No	Yes	Yes	Yes	No
Copying?	No	No	No	Yes	Yes	Yes	No
Collects circular structures?	Yes	No	Yes	Yes	Yes	Yes	Yes
Memory accessed	—	chunks which have references created or deleted	all chunks, twice	all chunks, twice	all chunks, most twice	young chunks, most twice	all chunks, once
Incremental form requires mutator check on chunk reference?	No	No	Yes	Yes	Yes	Yes	Yes
Other disadvantages	dangling pointers; memory leaks			doubles memory requirements	doubles memory requirements	doubles memory requirements	
Performance affected by:							
Program work	No	Yes	No	No	No	No	Yes
Chunk allocation	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Heap size	No	No	Yes	Yes	No	No	No
Used heap (bytes)	No	No	No	Yes	Yes	Yes	No
Used heap (chunks)	No	No	Yes	Yes	Yes	Yes	Yes
Relevant papers	(Nilsen, 1994; Nilsen, 1995)	(Deutsch & Bobrow, 1976; Corporaal *et al.*, 1990)	(Dijkstra *et al.*, 1978; McCarthy, 1960)	(Steele, 1975)	(Fenichel & Yochelson, 1969; Washabaugh & Kafura, 1990)	(Ungar, 1984; Tanaka *et al.*, 1994)	(Baker, 1992)

Table 2.1: Summary of garbage collection algorithms properties, see Section 2.4.7 for explanation

collection at these points could be expected to reclaim significantly more memory and/or be completed in significantly less time than a garbage collection performed at a random time (Kuechlin & Nevin, 1991).

Preventive garbage collection is most useful when used with a non-incremental algorithm, as this allows the entire garbage collection cycle to enjoy the optimal conditions.

In languages such as Smalltalk, where program state is commonly captured by saving the entire stack and heap, preventive garbage collection often used immediately before saving the heap. Immediately before saving represents a good time to perform a complete garbage collection.

2.7 Compile-time Garbage Collection

Compile-time garbage collection is a term applied to a group of program transformations applied at compile-time to reduce the complexity of the memory management for a system or subsystem. These include:

- converting temporary objects, which must be recreated on each entry into a scope, into static objects, which may be reinitialised.
- noting that certain objects have references to them created and deleted in certain narrowly constrained ways. For example the nodes used to construct a linked list may not be visible outside the list, and may be guaranteed to be garbage after the object they reference has been removed from the list. This is the case for example in the LinkedList class in appendix C.2.14, where such nodes are explicitly handled, because of the lack of compile-time garbage collection in the Java compiler I use.
- Delaying object allocation until the flow of control makes it unavoidable, to prevent the creation of temporary objects which are never referenced.

It may be noted that most of these are special cases and require a high degree of compiler sophistication to detect (Dean *et al.*, 1995; Mohnen, 1995).

Chapter 3

Programming languages and Garbage Collection

> "It is said that LISP programmers know that memory management is so important that it cannot be left to the users and C programmers know that memory management is so important that it cannot be left to the system." (Stroustrup, 1986)

The choice of a garbage collection algorithm for a system is intimately linked to the requirements of the language and memory model used (Finkel, 1995). The philosophy behind the language strongly affects the memory model used, whether the programmer considers memory management to be a system or a programmer responsibility and the relative difficulty of implementing garbage collection for the system.

3.1 Functional Languages

Garbage collection was invented for the functional language Lisp (McCarthy, 1960; Collins, 1960), and the overwhelming a majority of functional languages today (Lisp, Gofer, ML etc.) are garbage collected. Removing memory management decisions from the hands of the programmer, was seen as reducing the number of factors they had to be aware of when writing code. Garbage collection is such a part of the culture of functional programming, that most such languages are defined in such a way as to make garbage collection the only possible way of managing the memory. For example the definition of Scheme, a minimal dialect of Lisp says:

> "No Scheme object is ever destroyed. The reason that implementations do not [...] run out of storage is that they are permitted to reclaim the storage

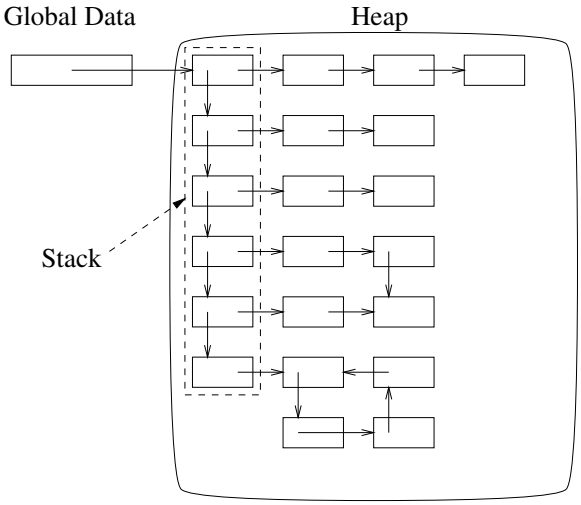


Figure 3.1: A very simple, Lisp-type, memory model

occupied by an object if they can that the prove object cannot possibly matter to any future computation." (Clinger & Rees, 1987)

Functional languages commonly have very simple memory models. For example Figure 3.1 shows one based on the 'cell,' a constant-sized object which contains two pointers or atoms and the type flags associated with them. Commonly, the execution-stack is implemented on the heap. This approach leads to the generation of large amounts of garbage, but eases the implementation of exotic control structures such as Schemes call-with-current-continuation. Such implementations rarely have low-level input/output facilities or other features requiring complex memory models, making garbage collection very straight-forward.

This simple memory model allows many optimisations and simplifications (Hamilton, 1995). The majority of garbage collection algorithms, such as reference counting, mark-and-sweep, semi-space copying, and the treadmill optimisation have emerged from functional language research.

3.2 Imperative Languages

Many traditional imperative languages, such as Pascal, Modula-2 and C, are defined, standardised and implemented without consideration of garbage collection. For this reason, there are often run-time structures such as untagged union fields, which are difficult or

impossible to type[1] and hard to collect.

Inability to type objects in memory leads to *conservative* garbage collectors, which are forced to over-estimate the number of pointers (Ganesan, 1994; Ferreira, 1991). These algorithms are unable to guarantee to collect all garbage, because constant data could have a value which, when interpreted as a pointer, points to a garbage chunk, and this chunk would never be collected.

More recent C++ garbage collection algorithms are *accurate*, being able to determine data from pointers, often using a memory and register partitioning system such as described in (Ganesan, 1994). This is typically requires restricting the language to disallow untypable objects. Some of these implementations also manage pointerful and non-pointerful memory separately (Ferreira, 1991).

In some applications, subsystems such as string manipulation are commonly implemented in a separate, non-garbage collected area of memory managed by customised, non-garbage collection, traditional memory management techniques. It is unclear which garbage collection algorithms perform best in these situation, but it appears that reference counting and mark-and-sweep algorithms are more amenable than are semi-space and generational algorithms to adaptations for this sort of use.

3.3 Object-Oriented Languages

Object-oriented languages such as Oberon and Eiffel are derivatives of the imperative line of languages[2].

Garbage collection algorithms for some object-oriented languages utilise a runtime system to type every object on the heap, to determine it's size, it's type and also which words are pointers and which are raw data. The overhead of such a system may seem prohibitive, but some object-oriented systems, such as Oberon already have a runtime system. Such systems, by providing necessary information at runtime, such as object type and size, have potential to support not only garbage collection, but also persistent objects (Mössenböck, 1993).

Some object-oriented systems, for example Smalltalk systems, generate vast quantities of garbage, such that (Ungar, 1984) estimates that their Berkeley Smalltalk system must reclaim 7/8 byte for every instruction executed. At the other end of the range, in most

[1] It should be noted that while languages such as Modula-2 are type-safe, this type-safety is enforced at compile-time, and type-data may be unavailable at run-time.

[2] there exist object-oriented functional systems, such as CLOS, which will not be considered here

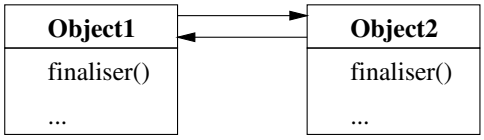


Figure 3.2: A finaliser ordering difficulty

Oberon systems, the overwhelming majority of objects are created on the stack and destroyed at almost no cost when the stack frame corresponding to the procedure to which they are local is popped (Heeb & Pfister, 1990?).

3.3.0.1 Finalisers

> "Destroy methods explicitly return the resource, making it available for reuse, or cause the deallocation of data structures. That is, they do what a garbage collector would do if it understood the resource." (Atkins & Nackman, 1988)

Object-oriented languages often include the concept of a 'finaliser,' a method called by the run-time system immediately before reclaiming the memory which the object occupies. Finalisers perform such actions as closing files and freeing non-memory resources. While finalisers present no problem for locally or globally allocated objects, finalisation of dynamic allocated objects is a problem, especially in a garbage collected system.

Ideally, the finaliser should be an arbitrary function specified by the programmer, with the full support of the language and run-time system. However, consider the situation in Figure 3.2 Object1 has a pointer to Object2, which has a pointer to Object1, neither object's finaliser can be invoked without potentially leaving the other with a dangling pointer.

In a language in which dynamic memory in managed by the application, resolution of this problem can be left to the application, since it controls the order in which objects are freed, and their finalisers called.

In a garbage collected language, there does appear to be a satisfactory solution for the ordering of finaliser invocation, however, several potential solutions are possible:

- exclude finalisers from the language, for example Oberon-2, thus allow for object initialisation but not for object finalisation (Mössenböck, 1993).
- allow finalisers to be called in an essentially arbitrary order, or one that breaches encapsulation. (For example Java, which invokes finalisers in inverse order of their

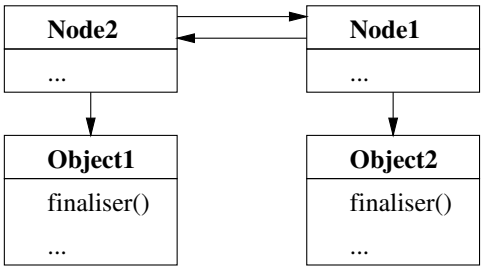


Figure 3.3: An alternative object-decomposition

initialisation (Sun, 1995).)

- prohibit finalisable objects from forming cycles, enabling an ordering to be imposed. Other than implying a fine-grain object-model, this is not as restrictive as it might seem. Figure 3.3 shows the objects in Figure 3.2 decomposed to fit into such a fine-grain object-model.

- Compile-time checks on the type of all objects potentially reachable from finalisers to build a partial ordering of classes. Queued finalisers are sorted in this order and then called from most general to least general.

- force the application to identify which pointers and references matter when determining finalisation order. For example the Boehm (Boehm *et al.*, 1991) collector, uses "disappearing links."

- prohibit finalisers from de-referencing dynamic pointers or references. Global variables may be safely used, since they are garbage collection roots, and only local variables are created during in invocation of the finaliser.

These solutions are language design issues rather than garbage collection issues, but they impact directly on the design and implementation of the garbage collector for the language.

Of the collectors I examined, only the Boehm (Boehm *et al.*, 1991; Boehm & Weiser, 1988; Boehm, 1993) and Java collectors dealt with languages allowing finalisation (C++ and Java), each dealt with ordering objects for finalisation differently. The Boehm collector used "disappearing links" – a list of pointers which were not to be considered pointers for the purposes of the finalisation ordering. It was assumed that all cyclic data structures would register sufficient disappearing links to make cycle transparent to the finaliser. The

Java collector finalises objects in the inverse order in which they were created—to quote the draft language specification (Sun, 1995) "finalisation should not be relied on for program correctness." The final language specification (Gosling *et al.*, 1996) omits this warning, describing instead an elaborate object life-cycle to allow for user invocation of finalisers and for finalisers to make other unreachable objects reachable.

One of the main uses of finalisation is the freeing of non-memory resources (for example file handles), and as there is no reference in the literature to any such resources which are inherently cyclical, it seems not unreasonable to limit destructors which cannot form cycles. This would greatly simplify finalisation, but also make it a much less general, making it less useful for debugging, for example.

If finalisation is being used to manage non-local resources, it may be desirable for finalisers to be run on all objects at system termination, even if the termination is due to an exception being thrown.

3.4 Concurrent Languages

Considerable work has gone into garbage collection in concurrent and parallel systems, (Harbaugh & Wavering, 1991) examines the work described in (Baker, 1978) (see Section 2.5), and shows that incremental garbage collection can be implemented concurrently, especially when it can be run on a dedicated CPU in the address space as the mutator.

In (Kuechlin & Nevin, 1991), the authors present a complex threaded system which uses local and global garbage collections to efficiently handle memory management.

(Kuo & Kuo, 1993) present a distributed scheme based entirely on preventive garbage collection. The application program is a purely functional array processor which works by delegation with each node on the expression tree being computed by a separate node. When a server has performed it's computation and returned it's result the node's entire heap is collected.

(Boehm *et al.*, 1991; Kordale *et al.*, 1993; Tanaka *et al.*, 1994) represent another line of research, using parallel mark-and-sweep collectors and the virtual memory hardware of stock computers to trap chuck writes during marking. The performance of this type of collector differs significantly from those based on Baker's work:

- While they may be guaranteed to collect all garbage during a collection, it appears uncertain whether they can be guaranteed to collect garbage faster than a pathological mutator could generate it.

- Because they are non-copying, references into a mark-and-sweep maintained area of memory remain stable for long periods of time, avoiding indirection tables when references are saved to files, passed across a network or otherwise passed beyond the direct control of the collector.

Baker's write-barrier (which marks all objects with newly-created references to them grey), is used extensively in concurrent systems, since it allows fine-grained concurrency (both within the application and within the garbage collector) without locking.

3.5 Real-Time Languages

Real-Time applications are those designed to produce correct results while meeting predefined deadlines (Panzieri & Davoli, 1993). Some authors use "real-time" garbage collection to mean "on-the-fly" garbage collection—garbage collection is done without taking the application off-line. (Kuo & Kuo, 1993) and (Tanaka *et al.*, 1994) both present implementations which meet this modified definition of real-time.

True real-time performance is not achieved by building programs that run fast, but by building programs which perform actions in a guaranteed (short) period of time (Baker, 1978; Washabaugh & Kafura, 1990; Corporaal, 1991a; Corporaal, 1991b). In general there is a tradeoff between selecting an algorithm with the best average performance and selecting an algorithm with the best worse case performance. The Baker algorithm, for example, can be tuned so that it only just copies the semi space before flipping, thus minimising the overhead of allocations, but incurring a much higher probability of indirect referencing each time a chunk is referenced, increased the total overhead of garbage collection to the system.

It may be noted that very similar tradeoffs occur in traditional memory management systems. (Nilsen, 1995; Gao & Nilsen, 1994) presents a number of such systems, comparing them to the authors' garbage collection algorithm, each of which has distinct average- and worse-case time for allocation and reclamation.

Interactive applications are not necessarily real-time. Interactive applications typically measure garbage collection in terms of user-responsiveness — 'does an increment of garbage collection while the user is mousing cause undue user-disorientation ?' — and the answers to such questions derived from user-testing. Real-time applications typically measure garbage collection in terms of calculated worst-case upper-bounds of operations.

An additional problem occurs with object-oriented, real-time languages, in that finalisers, which may contain arbitrary code, may take arbitrarily long to execute. There appear

to be two solutions to this, raise the upper-bound of the increment or, in a multi-threaded system, have the finalisers invoked from a separate thread

Hard real-time garbage collection can be achieved, as illustrated by (Magnusson & Henriksson, 1995b).

3.6 Distributed Languages

Distributed systems, pose a problem for garbage collection in that copying garbage collectors invalidate all external references to chunks when they are copied. For this reason, many modern distributed systems use mark-and-sweep (Tanaka *et al.*, 1994; Kordale *et al.*, 1993) or even reference counting (Lins, 1992; Corporaal *et al.*, 1990; Jones & Tyas, 1994), with custom modifications to collect circular structures.

There are two key features which cause problems in distributed systems:

1. cyclic data structures in which the cycles span multiple machines
2. widely distributed, deep data structures

I know of no collectors able to handle these two features without global synchronisation points of all threads on all nodes, effectively taking the system off-line for garbage collection. These affect both message passing multiple address space systems and single address space systems, as the difficulties lie in the object-connectivity model and essentially arbitrary run-time connectivity of objects.

A good survey of of distributed garbage collection can be found in (Plainfossé & Shapiro, 1995).

Chapter 4

Object-Orientation, Abstraction and Design Patterns

Abstraction. There should be a way to factor out recurring patterns. (Finkel, 1995)

4.1 Object-Orientation

Object-orientation is a software design technique in which each entity is modelled by an *object*. Objects may represent a real world entity (e.g. a lily), an aspect of the computer (e.g. a hard disk), a logical value (e.g. darkness) or an abstract construct (e.g. a mathematical theorem). Objects have both state (internal variables) and methods (valid manipulations of the object). The state of an object may be neither examined nor changed except through the object's methods, the state is said to be *encapsulated*.

For example, a Lily object might have colour and picked variables, and observe() and pick() methods[1]. colour is a constant *variable*[2]—once the object has been created, it's colour may not be changed. picked, a variable which indicates the health of the Lily, is very readily open to change—pick()ing the Lily will kill it! The change in state is limited, however, to that allowed by the two methods observe() and pick(). There is not a way to reach inside a pick()ed Lily and make it live again. Similarly, repeatedly pick()ing a picked Lily does not restore it to life. As might be expected, observe() allows an observer to see the colour of the Lily, and whether it has been pikced.

It will soon be realised that one Lily is much like the other. Defining the concept of

[1] in reality, such an object would undoubtedly have considerably more variables and methods, but I shall only consider a few, for the sake of brevity

[2] note that variable is used as a unit of state, and does not necessarily indicate that the value stored is open to change.

Figure 4.1: Class diagram for the class Lily

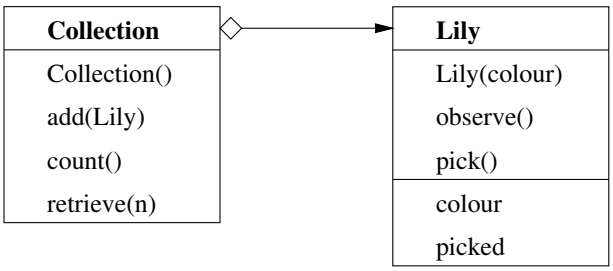


Figure 4.2: Class diagrams for Lily and Collection, showing their connection

Lily each time I encounter a new individual (or *instance*) would be a waste of effort. On the other hand, if I define a *class* Lily, then each time I encounter a new instance, it is sufficient to supply it's relevant attributes (in this case it's colour). I already know what methods it has (observe() and pick()). I need a method to construct these new instances, which I give the same name as the class (e.g. Lily()).

Object-orientation provides a diagramatical description for defining a class, called a *class diagram*, the class diagram for Lily is shown in figure 4.1. The name of the class is shown in bold at the top, followed by the methods and finally the state variables. If the state is considered unimportant, it is omitted, as are the methods for very common, or already discussed, objects.

Now I have defined a class Lily, I can make instances of the class, observe() them, assign them to variables, pick() them and so on. However, if I are keen on them and wish to collect a significant number of them, keeping track of them all is going to be problematic. The solution to this is to create a Collection of Lilys.

Figure 4.2 shows class diagrams for Lily and Collection, and the relationship between

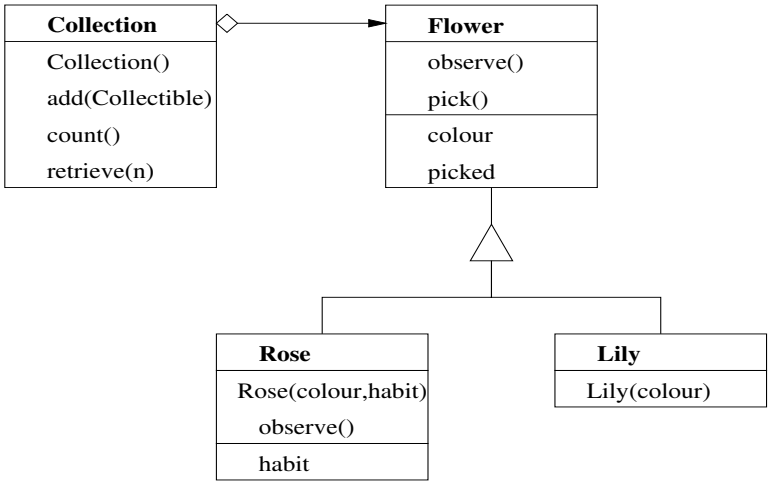


Figure 4.3: Class diagram for Flower, Lily Rose and Collection

them. ◇→ is read "is a collection of," so a Collection is a collection of Lilys. A Collection has a constructor and three other methods: add(Lily) which adds a given Lily to the Collection, count() which returns the size of the Collection and retrieve(n) which returns the n'th Lily in the collection. No state is specified for Collection, because it is not directly relevant to the task at hand. There is also a need to "hide" or encapsulate the data—if no other parts of the system are aware of Collection state, they can not be broken if the internal workings are changed. For example, if the Collection got to big to fit into main memory, I might have to change Collection to store it's state on disk. Provided I maintain the same interface (methods) I can do this without adversely effecting other parts of the system.

While are out expanding my Collection, I may notice other types of flower in the garden, and wish to collect them also. It would seem sensible to create Rose objects and add() them to the Collection. This would cause two problems. Firstly, I have specified only a method to an a Lily to the Collection. Secondly, roses are known to come in different habits (bush, climbing, standard etc), and I want to capture this in the Rose objects. The solution is *inheritance*, which captures the common details in a class called Flower, and deriving the various type of flowers as sub-classes of it (see Figure 4.3).

The symbol —◁ is read 'inherits from'—Lily and Rose inherit from Flower. Both methods and state are inherited, so pick() needs only be defined once, since pick()ing a Lily or a Rose have the the same effect. However, Rose has *overridden* observe, since someone looking at a Rose needs to know both it's colour and it's habit. Note that Flower

is not provided with a constructor, this prevents the creation of a Flowers which isn't also a Lily or a Rose.

4.2 Abstraction

Abstraction is grouping similar 'things' together and naming them. For an abstraction to be of use, it needs to have a effective definition, a rule or set of rules by which any 'thing' can be judged to be, or nor to be, a member of the group. The name is used to refer to both the group of 'things' and their defining properties.

Abstraction hides unimportant detail while highlighting important information. Obviously, the definition of 'important' is dependent on the context and attributes under consideration. For example, the details of the implementation of Collection are irrelevant, as long as it's methods work.

The notion of class is an abstraction method for grouping individual objects. Thus if all flowers have commonality (colour, pickedness etc), they are captured in the class Flower.

The notion of inheritance is an abstraction method for grouping classes. When I wanted different flowers to become part of the Collection I captured their commonality in Flower.

4.3 Design Patterns

Design patterns (Gamma *et al.*, 1995) are an abstraction method for use with groups of objects to capture interactions between the objects.

They are chiefly used as tools for documenting design decisions. Formerly called frameworks, they are descriptions of groups of objects and classes used to solve a common design problem in a specific application domain or context. A design pattern captures the framework within which the development, and reuse (Rada, 1995), of a specific type of software component (or group of components) occurs. Probably the most well-known example of a design pattern is the Model/View/Controller used in Smalltalk to develop user interfaces (Buschmann *et al.*, 1996).

By drawing on catalogues of previously encountered, described and documented design patterns, it is hoped that future designs will be eased by reusing the solutions to previous problems (Gamma *et al.*, 1993). Such a catalogue needs to concisely capture the situations in which the design pattern is found, the problem it solves within that context, and how it solves the problem.

Object theory suggests that all parts of a system are amenable to object-oriented design, and having these design decisions captured, from the high-level user-interface components to the low-level kernel components.

Design pattern theory suggests that there are both patterns common to widely disparate systems (e.g. both a text editor and a robot control system might use Interpreters (Gamma *et al.*, 1995)) and patterns restricted to a narrow class of applications (e.g. two robot control systems might use one or more design patterns particular to the field of robot control).

A group of design patterns, which together express an approach to a broader problem, constitute a pattern language. Such groupings of patterns by domain of application seem particularly popular in low-level or operating system design, and several for multi-processor systems having been published recently (Aarsten *et al.*, 1996; McKenney, 1996), and one for porting micro-kernels (de Champlain, 1996a). The intent of a pattern language is to capture as much as possible of the domain knowledge in the area and make explicit the effects of this knowledge on the design decisions in the systems built for this domain.

Larger catalogues of patterns for more general use, such as (Gamma *et al.*, 1995), are arranged according to the aspects of object dealt with. Creational patterns deal with object instantiation, structural patterns deal with the way classes and objects are composed to form larger structures, and so on. The intent is to guide the user of such a collection to the pattern or patterns within the collection which are of use for the task at hand. Many such classification schemes of patterns are possible, including a layered approach and as well as those based on 'communicates with' and 'incorporates' relationships (Zimmer, 1995; Buschmann & Meunier, 1995).

It is anticipated that once design patterns have been in use for a period of time, catalogues will appear documenting and classifying commonly used and widely known patterns and pattern languages. These may nor may not be similar in nature to catalogues of abstract data types have have previously occurred (Uhl & Schmid, 1990). System designers will the be able to refer to these patterns when designing and/or describing a system, and tool designers will be able to develop design and implementation environments based on these (Pagel & Winter, 1996).

Another product which will hopefully emerge eventually is a better view of strong points and weaknesses in the design pattern approach to system design. While critiques of the design patterns have been published, for example (Cline, 1996), a lack of hard data on design patterns in the real world is hampering impartial evaluation. Such evaluation is only likely once design patterns have passed through the current 'evangelistic' phase and

been used extensively in the real world.

4.4 Describing design patterns

In the literature the descriptions of design patterns range from the anecdotal (Gabriel, 1996) and informal (Coplien & Schmidt, 1995) to the semi-formally (Gamma *et al.*, 1995). In the future, it may be hoped that formal descriptions will appear so they can be used in a semi-automated, tool-driven manner.

When describing new, or cataloguing old design patterns, a uniform description method is important, since users need to identify and compare patterns to be able to quickly decide whether a particular pattern fits their need. Commonly, design patterns are described in terms of:

Name A unique name for the pattern

Context A typical context for the pattern's use, including a brief description of the domain of applicability

Problem What particular design issue or problem does it address? For what (its purpose)?

Solution How the problem is solved

Applicability When to use the pattern. What are the situations in which it can be applied? (Any particular exclusions should be noted)

Consequence What are the consequences, flow-on effects down-sides ?

Implementation An example of the way the pattern is used in practise, in an appropriate language.

These categories aim to capture as many of the design decisions and implications as possible, to make explicit the trade-offs, and to suggest possible alternatives to a decision.

4.5 An example design pattern

In the Rose example earlier in the chapter, I had a single Collection, and all Flowers were placed in in when they were collected. When I am in a position to guarantee that there will only be one of an object (e.g. a Collection), this object is called a Singleton. Other common Singletons include print spoolers, process tables and system clocks.

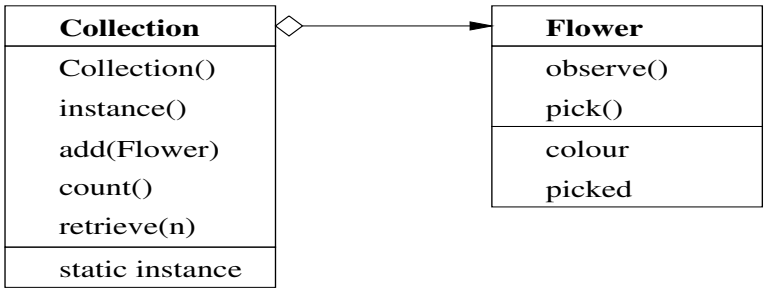


Figure 4.4: Class diagram for Lily and Collection showing the Singleton nature of Collection.

Name Singleton

Context Found across a broad range of systems

Problem For some classes it is important for some classes to have exactly one instance. It's necessary to ensure that only one instance exists, and is readily available

Solution Have the static reference to the single instance within the class, which can be accessed globally, and a guard condition in the constructor to prevent the creation of multiple instances of the class.

Applicability Objects which are conceptually, as well as implementational unique.

Consequence A system built on the assumption that instance() is available is likely to be hard to modify to account for multiple instances of the class. Thus it's imperative that the singleton is both conceptually and implementationally unique (e.g. changing the implementation to allow for several Collections (by other floral lovers perhaps), would be difficult). Singletons have the advantage over global variables (another common solution to this problem) both in terms of namespace conservation and sub-classing ability.

Implementation The following code implements the Singleton Collection and the rest of the extended example used throughout this chapter. The Vector being used is a general purpose list/random access data structure present in the standard libraries for Java and assumed to be reasonably efficient. Because Java lacks parametric types Vector is defined in terms of Object and extracted objects then cast to Flower.

```
/**
 * The abstract class Flower, which collects the common aspects of Rose and Lily
 */

class Flower {
  String colour;
  boolean picked;
  // other state ...

  public String observe(){
    return colour;
  }

  public void pick(){
    picked = false;
  }
}

/**
 * The Rose class
 */

class Rose extends Flower {
  String habit;

  public Rose(String colour, String habit){
    this.colour = colour;
    this.habit = habit;
  }

  public String observe(){
    return colour + habit;
  }
}

/**
 * The Lily class
 */

class Lily extends Flower {
  public Lily(String colour){
    this.colour = colour;
  }
```

```
}

/**
 * The Singleton class Collection, which is a container of Flowers
 */
                                                                    50
class Collection {
  private java.util.Vector flowers;  // a place to store the Flowers
  private static boolean instantiated = false;
  private static Collection instance = new Collection();

  public Collection(){
    if (instantiated == true) // guard against multiple instantiation
      System.out.println("Error: attempt to create a second Collection");
    else {
      flowers = new java.util.Vector();                             60
      instantiated = true;
    }
  }

  public Collection instance(){
    return instance;
  }

  public void add(Flower flower){
    flowers.addElement(flower);                                     70
  }

  public int count(){
    return flowers.size();
  }

  public Flower retrieve(int n){ // cast the returned element back to a flower
    return (Flower) flowers.elementAt(n);
  }
}                                                                   80
```

Chapter 5

Object-Oriented Analyses

> A design pattern packages expert knowledge; it represents a solution to a common design problem and can be reused frequently and easily. Each pattern is a micro-architecture on a higher abstraction level than classes (Krämer & Prechelt, 1996).

I have examined a number of garbage collectors, each either described in the literature or available as source code, principally the Tolpin, Boehm, Baker78 and Java collectors. Each of the collectors represents a very different implementation with different objectives. The Tolpin collector is a non-incremental mark-and-sweep collector, part of the run-time system for an Oberon-to-C translator, with access to complete type information. The Boehm collector is a general purpose collector for C/C++ with no access to type information. The Baker78 collector is an incremental collector for Lisp, and relies heavily on the traditional Lisp type system. The Java collector is part of run-time system in Sun Microsystems Java Developers Kit, (May 1995 release), it's an abortable, non-incremental, mark-and-sweep collector with access to complete type information.

While none of the collectors is modelled, described or implemented using object-oriented methodologies, I have attempted to perform an object-oriented re-design of the collectors. The resulting schemata were then examine for similarities, design patterns.

The Baker78 collector is written in 'pseudo-Algol-BCPL' (Baker, 1978) while the other three collectors are written in C (ISO9899, 1990). The Boehm collector has an additional adaptor written in C++.

For each of the collectors, I shall first briefly describe the collector, and then describe it's important features — those features which differentiate it both from the other garbage collectors described here and those encountered in the literature. Finally for each, i shall then give an object-oriented redesign using standard notations and diagrams.

5.1 The Baker78 Collector

The Baker collector, described in detail in (Baker, 1978) is an early implementation of a real-time, incremental garbage collector for a Lisp run-time system. It's extreme simplicity can be traced to the use of a 'cell' (see Section 3.1), and to the simplicity of the algorithm which is a textbook example of a semi-space copying algorithm (see Section 2.4.4).

5.1.1 Important Features

1. Synchronisation is achieved thought a write barrier. This detects writes to objects (changes in their state) during garbage collection and forces the object to be trace()d before the write may continue. Thus the penalty for the write barrier is distributed across writes during the collection period.

2. The ordering of objects on the heap is used as the order for tracing. This results in a *breadth-first* traversal of the heap, and minimal overhead for additional data structures.

3. The collector is type safe and accurate, i.e. it knows whether each address in memory holds either a pointer or data and uses this to ensure that only those objects which have references to them are copied.

5.1.2 Object-Oriented Redesign

Figure 5.1 shows the class diagram for the Baker78 collector. Note that the memory model has a trivial root, as shown in figure 3.1.

StackIterator is an iterator acting on the subset of ObjectProxys which make up the program stack. ObjectIterator is an iterator acting on ObjectProxys. ObjectProxy is a proxy for Object, which makes sure that the application always references the copy of the Object in the correct space.

Figure 5.2 shows a trace of object interactions when the Application calls new(). trace() is called in each of ObjectIterator and StackIterator to perform an increment of garbage collection, before the new Object and ObjectProxy are created by Space.

Figure 5.3 shows detail of the Collectors call to ObjectIterator, showing the creation of a new Object, the reading, translating and writing of data from the old Object to the new Object. The old Object is then updated to point to the new Object, which is marked traced() and add()ed.

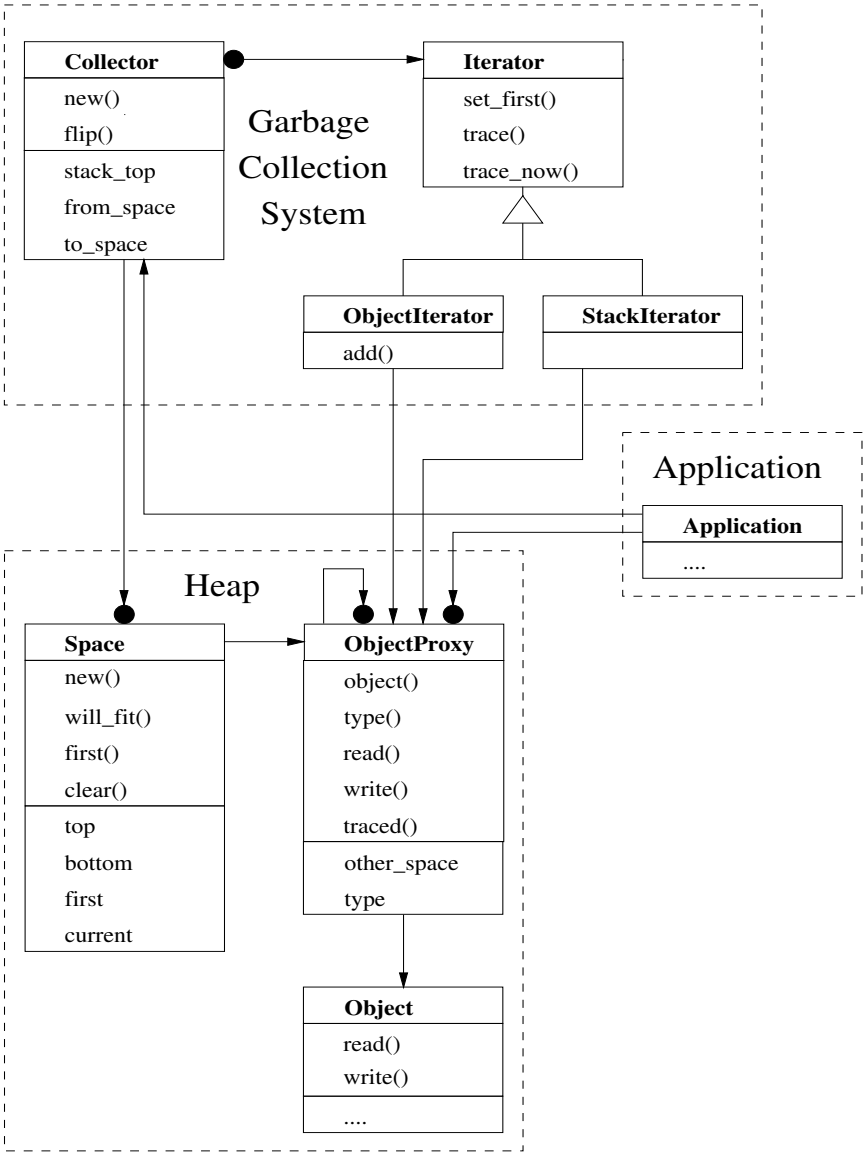


Figure 5.1: Class Diagram for the Baker78 collector

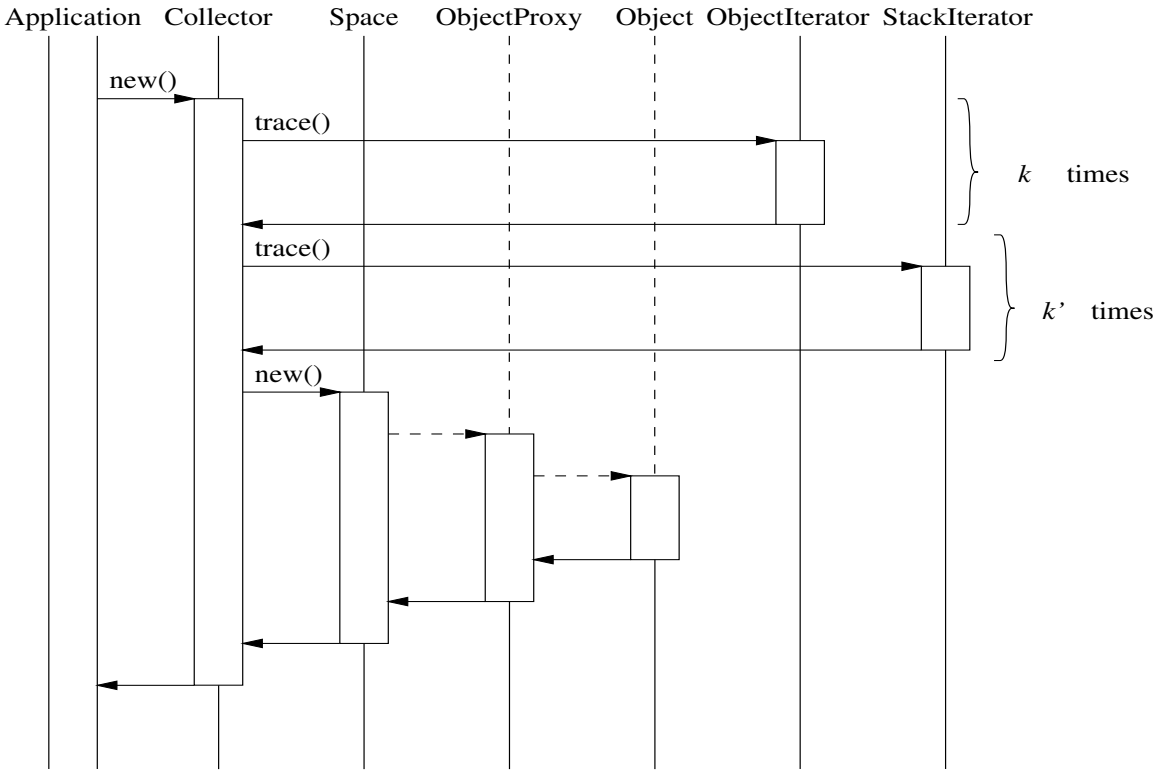


Figure 5.2: An interaction diagram a new() call. Both k, the number of heap objects traced per allocation, and k', the number of stack objects traced per allocation, are typically 2. Tracing 2 objects per allocation ensures that at most 1/2 the

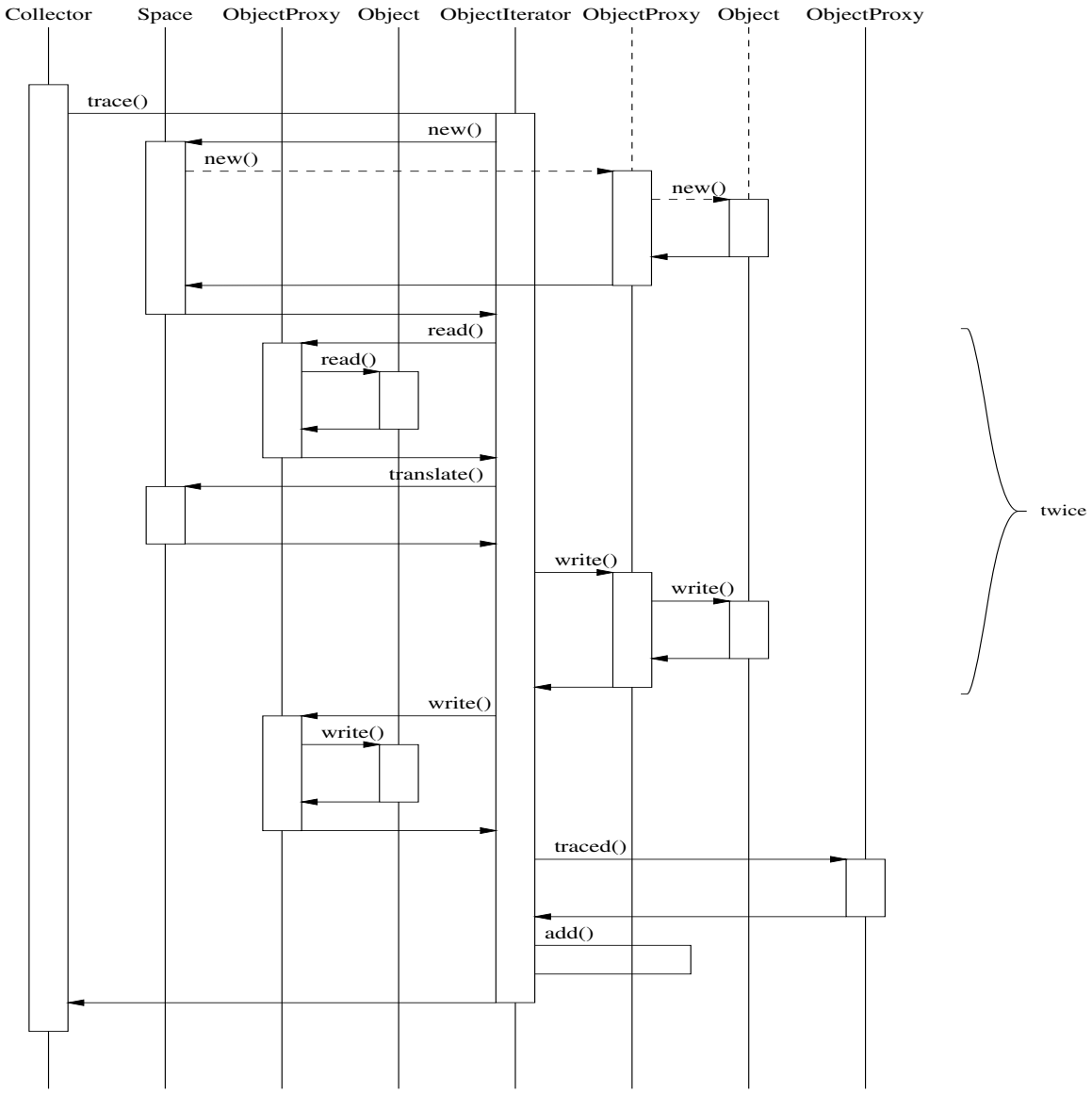


Figure 5.3: An interaction diagram of a trace() call to ObjectIterator. The central section is repeated twice, once for the cdr and once for the car of the object.

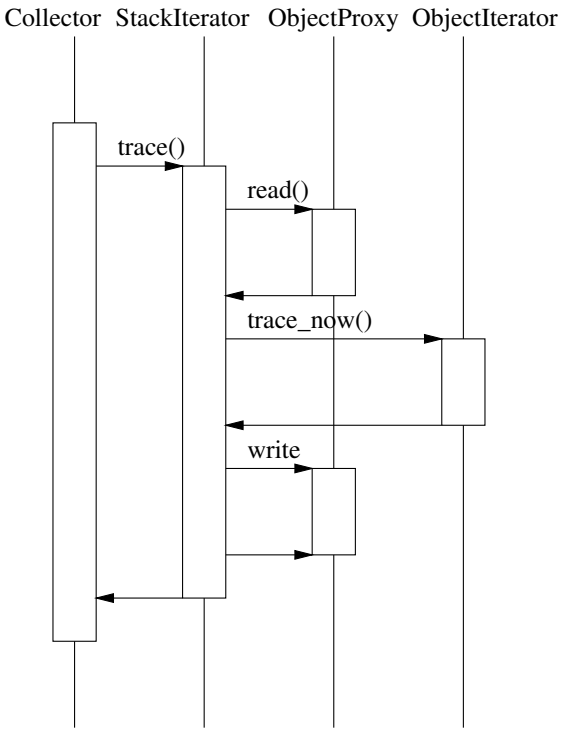


Figure 5.4: An interaction diagram of a trace() call to StackIterator. The StackIterator is smaller than ObjectIterator for two reasons, firstly because it connectivity of the stack on the heap, it knows that

5.2 The Tolpin Collector

pOt is an Oberon (Wirth & Gutknecht, 1990) to C (Kernighan & Ritchie, 1988) translator written by David Tolpin (dvd@Jet.Msk.SU). As part of it's runtime library it includes a garbage collector, hereafter referred to as the Tolpin collector. The Tolpin collector has access to the runtime system, so it always knows the type of every chunk. This allows accurate collection—all chunks which survive a collection were reachable at some stage during the collection. The collection is non-incremental, with collections being triggered either a) automatically after allocation of a certain number of bytes on the heap or a failed call to malloc or b) explicitly by the application[1]. The non-incremental nature avoids the need for a write block as is required in both the Baker78 and the Boehm systems.

The allocator could very easily be made thread-safe, but, because it lacks a read or write barrier, garbage collection would still be non-incremental.

The allocation algorithm used is very simple

1. malloc() the chunk of memory from the operating system
2. perform a full garbage collection if either the malloc() failed or a certain number of bytes have been allocated since the last garbage collection
3. register the chuck (add a reference to it to the Heap).

The collection algorithm used is also very simple

1. scan the stack, removing every chunk referenced from the Heap and adding it to the MarkedHeap, in a depth first manner
2. free() all chunks remaining on the heap list, as they are not reachable.
3. switch the names of the Heap and the MarkedHeap[2].

5.2.1 Important Features

1. The design shows a preference for simplicity over efficiency. The three central data structures are all implemented as lists, even though two of them (Heap and MarkedHeap), could be much more efficiently implemented as binary trees or similar structures. I *estimate* that the 'inner loop' of the Marker, markptr(), is order $\frac{1}{2}mn^2$

[1] The application the Tolpin supports being the rest of the runtime library, *NOT* the code being translated.

[2] This aspect of the collector is similar to Baker78, in that chunks (or references to them) are moved from one (area) list to another as they are traced, and those chunks not moved are considered garbage.

(where n is the number of chunks on the heap and m is average number of pointers per chunk), and that this could be reduced to order $mn \log(n)$ if the implementation of heap was changed to a sorted, balanced, binary tree.

2. Freelists are managed externally. Memory from garbage chunks is freed, and then later malloced when new chunks are allocated. This gives the potential for returning memory to the operating system, a major bonus when running in a non-steady state on a shared platform.

5.2.2 Object-Oriented Redesign

The data structures are very simple:

1. a linked list of references to every stack frame (including the main frame, modules and global frames).

2. two linked list of references to heap chunks, the heap and markedheap.

5.3 The Boehm Collector

The Boehm collector is an highly-evolved, conservative, incremental mark-and-sweep garbage collector. It assumes to no access to type information, but uses standard UNIX virtual memory (VM) hardware. Synchronisation is achieved thought a write barrier. This barrier detects writes to object by checking the 'dirty bits' of VM pages after tracing. Thus the penalty for the write barrier is incurred at the end of the marking phase of the collection. Objects are traced using a separate stack, resulting in a *depth-first* traversal of the heap, and overhead for additional data structures. Explicitly created 'stubborn' and 'uncollectable' objects are not repeatedly scanned and considered for reclaiming respectively.

C++ is non type-safe language, forcing the collector to be inaccurate, that is, it is uncertain which memory locations hold pointers and which hold data. In some cases bit patterns in data, which would be pointers into heap if they were pointers, may cause incorrect memory retention. Blacklisting is used to combat this, but doesn't prevent all incorrect memory retention.

Unlike the Baker78 and Tolpin collectors, the Boehm collector is designed using abstract data types and clean separation between subsystems.

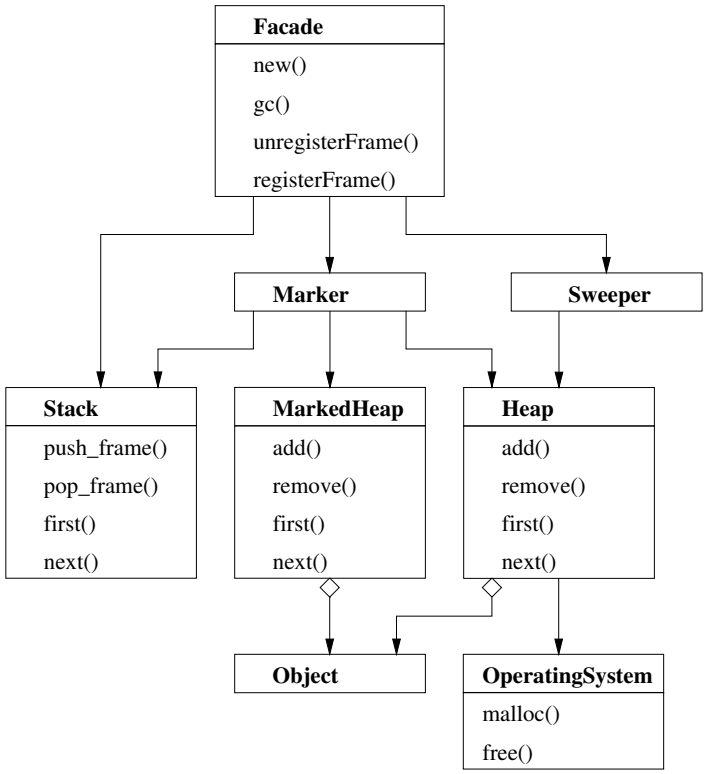


Figure 5.5: An object model for the Tolpin collector, showing application and operating system interface. Note that in this case the application is the rest of the runtime library. All objects are assumed to have access to runtime library type information.

5.3.1 Basic algorithm

Marker scans the program stack, adding Object references there to the MarkStack. Each Object on the MarkStack is then examined for object references and referenced Objects added to the MarkStack if unmarked. When the mark stack is empty, the entire heap has been traversed.

The DirtyHandler then places all unmarked Objects on dirty memory pages on the MarkStack and examines them, in much the same way that Marker did.

Sweeper then sweeps the entire heap, using the HeapSectorsList to ensure it examines every Object, and places references to all unmarked Objects on the ReclaimList.

Reclaimer then calls the finalisation function, if any, for each Object on the ReclaimList, and adds them to the Freelist for the appropriate size of Object.

The action in all four phases of the collection is incremental.

5.3.2 Important Features

1. Extra roots can be added to the system (for example by the threads package), these are scanned and object references added to the mark stack before the marker is called.

2. The threads object provides an interface to the Solaris threads package. The stack for each thread is automatically added to the roots. In a multi-threaded situation, locking is used for exclusive access to shared data structures. Signals are explicitly disabled by the dirty page handler while it places all unmarked objects on dirty memory pages in the MarkStack.

3. Because of the way the dirty page handler works, objects allocated during the operation of the marker (the longest phase of the collection) are eligible for immediate collection if they promptly become unreachable.

4. If the garbage collector is working in incremental mode, each allocation causes a small amount of collection to be performed. In non-incremental mode, a complete garbage collection occurs when the heap is exhausted. In either incremental or non-incremental modes a full garbage collection may be called at any time by the application.

5. Blacklisting is a technique for detecting potential false 'hits' introduced by conservatism, causes holes to appear in the heap. This increases the total memory used

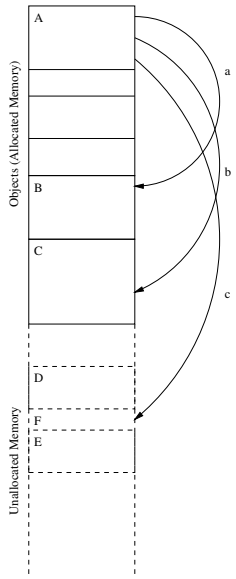


Figure 5.6: Pointers and application data viewed as pointers. (see Important Feature 5)

by the system, but reduces the probability of an unbounded memory leak due to conservatism.

If A (see Figure 5.6) is an chunk which contains a pointer 'a' and some application data bytes 'b' and 'c'. 'a' points to chunk B, the garbage collector, as it traces A finds 'a', correctly follows it to B and ensures that B is also traced. 'b', however, is also also viewed by the garbage collector as a pointer, and incorrectly followed to chunk C, leading to the retention of C (and all chunks referenced by C) even if the application no longer needs these chunks. The collector knows that 'c' is not a pointer due to the fact that if it were, it would be invalid (point to memory not yet allocated). Whenever it comes across such a piece of application data which is a 'close miss' (would be a pointer to memory soon to be allocated, if it were a pointer), it places them on a Blacklist. New chunks (D and E) are placed in memory to avoid such blacklisted locations. This increases fragmentation, F remains unallocated. Because application data changes, blacklists are aged or 'promoted' in such a way that F may be allocated at a later time, if application data 'c' changes.

Blacklisting, and indeed retention, can be expected to occur in proportion to the occupancy of the address space, since the if a high proportion of the address space is used there is a high chance of "random" application data looking like a pointer

to an chunk. Thus an application which filled $\frac{1}{2}$ a flat 16 bit address space could be expected to have a $\frac{1}{2}$ chance of each word being interpreted by the garbage collector as a valid pointer. With the same application in a 32 bit address space as would have a $\frac{1}{2} \times \frac{1}{2^{16}}$ chance. In a 64 bit address space, the chances of false hits would be vanishingly small. Unfortunately, application data isn't truly random, nor evenly distributed, so this approximation breaks down at the limit.

6. If the application is aware of the garbage collector, it may request objects which are handled in specific ways:

   normal

   atomic —guaranteed by the application not to contain pointers, and never scanned for pointers by the collector

   stubborn —guaranteed by the application not to change without appropriate calls to startStubbornChange() and endStubbornChange(), thus reducing the work required of the write-barrier

   uncollectable —guaranteed by the collector never to be collected

   ignoreOffPage —the application guarantees that if an ignoreOffPage object is live, there is a pointer to it's first page. This eases fragmentation problems caused by the blacklist when allocating large objects, such as images.

This widening of the interface between the application and garbage collection incurs the cost of deviation from the standard interface (ISO9899, 1990), binding the application more tightly to the collector and making it less portable.

5.3.3 Object-Oriented Redesign

Figure 5.7 gives my interpretation of an overview of the Boehm garbage collector. Note that coloured objects (□) are those with which the application interfaces.

The internal data structures consist of a number of well defined, narrow-interface, data structures. The Roots, MarkStack and ReclaimList are used to store the collection roots, the state of the heap traversal and the objects to be reclaimed, respectively. The Freelist is an an array of lists of free chunks, each list holding chunks of size n. The HeapSectorsList is a list of those portions of the heap which are valid; by removing the restriction that the heap be contiguous, HeapSectorsList allows the collector to handle dynamically-linked libraries, memory mapped files and similar 'features' of UNIX memory.

The BlackList is a narrow-interface to the data structures to implement blacklisting. Four data structures are maintained, two for the heap and two for the stack. At any point

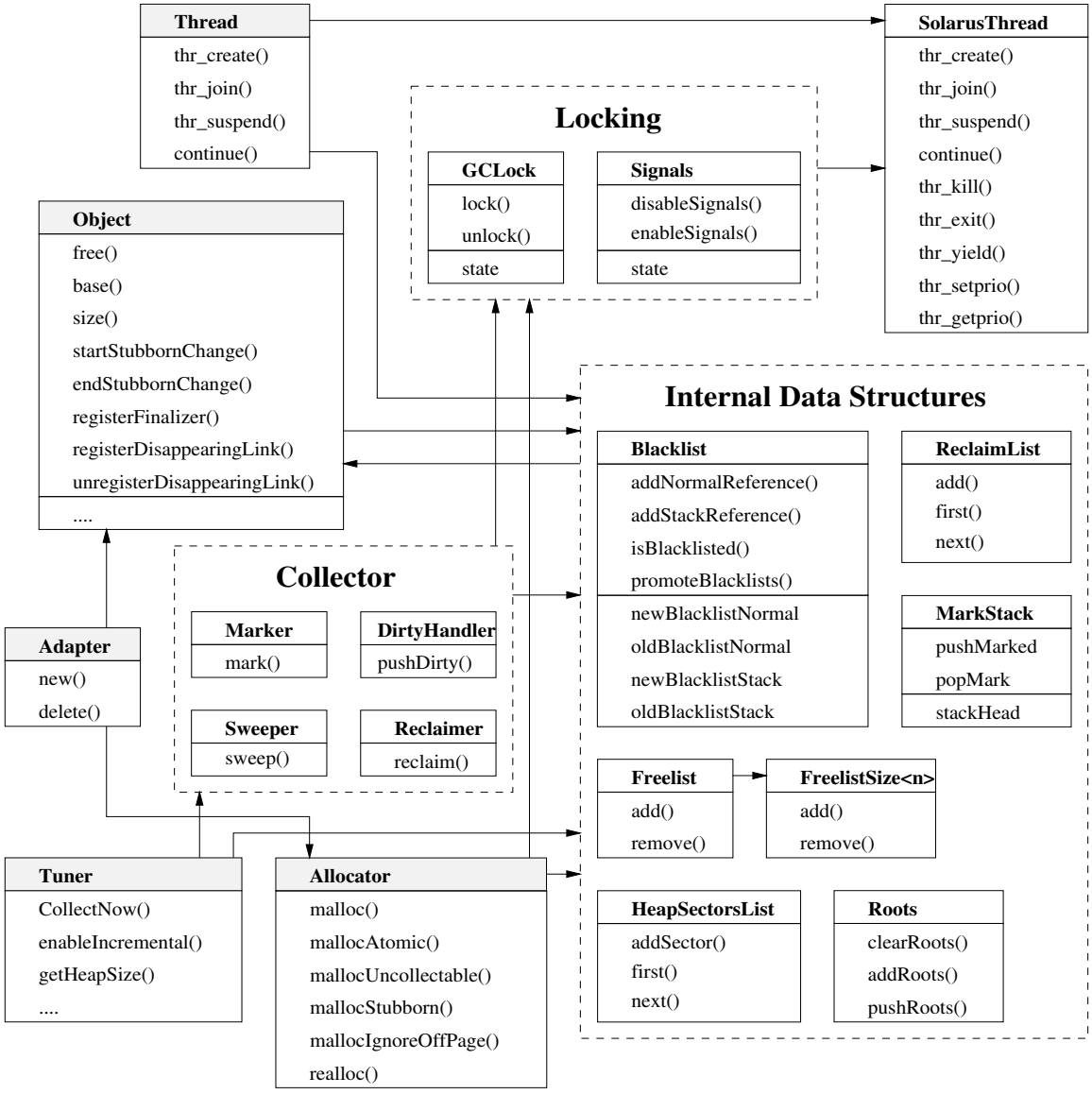


Figure 5.7: An object decomposition diagram of the Boehm collector. Note that duplicate data structures for alternative Object types.

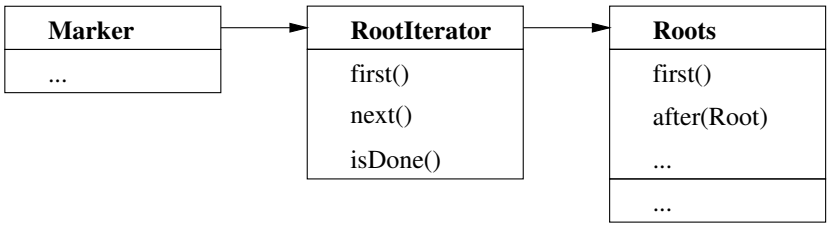


Figure 5.8: An example of a utility class, RootIterator, as used in the Boehm collector

in the collection, the 'old' lists are in use for blacklisting, while the 'new' lists are being built to replace the old and the end of the cycle.

Mutual exclusion is achieved in classic UNIX style, using a single Lock for the heap data structures and simple Signal manipulations (Tanenbaum, 1992). A simple thread-awareness is provided for multi-threaded applications which wish to have a low-priority garbage collection Thread. Currently the package only supports SolarisThreads (SunMan, 1995).

The Allocator is an interface which provides the fundamental interface to the system, allocation of chunks and the automatic triggering of garbage collection. The Tuner provides access to a more sophisticated interface for collector-aware applications.

An Adaptor provides a C++ interface to enable the collector to be used for pure C++ programs. The new() is converted directly to a normal malloc(), delete() can be configured to overwrite an object with zeros. Such an overwriting reduces the conservative error, but introduces the possibility of incorrect deletion resulting in applications dereferencing null pointers (crashing), such methods are useful when detecting memory leaks.

The Object is a proxy which provides both an interface to finalisation and stubbornness for garbage collection for collector-aware applications and a receptacle for per-object data (chunk size etc).

In addition to those shown in Figure 5.7, there are a number of utility classes, such as RootIterator, which is shown in Figure 5.8.

5.4 The Java Collector

The Java garbage collector is considerably different to the other garbage collectors considered so far. It is a non-incremental, interruptible, compacting garbage collector.

The Java collector is a non-incremental collector—a complete garbage collection, and

possibly compaction is performed in a single unit. If the collection is being performed during idle time, a pending interrupt will terminate the collection process. A collection performed on memory exhaustion is neither interrupted nor terminated by a pending interrupt.

Compaction of the heap is only performed when the heap does not contain a large enough free block to satisfy an allocation (a special case of memory exhaustion).

Finalisation is another unusual aspect of this collector (which it shares with the Boehm collector). Objects may request finalisation by placing themselves on the hasFinaliser queue, and are guaranteed to be finalised before objects they reference. No other guarantees are made about the timing of finalisation.

5.4.1 Algorithm

Figure 5.9 shows the objects involved in garbage collector in the Java runtime system from Sun Microsystems. Coloured objects are global container objects which the collector has access to. All Objects are accessed via a Handle which is a *proxy* controlling access to the Object.

Garbage collection can be initiated in either of two ways. Firstly, if the system has been idle for more than a second, an asynchronous, interruptible, garbage collection is performed by the garbage collection thread. Alternatively, if memory is exhausted, a non-interruptible, synchronous garbage collection is performed.

The first step of garbage collection is to zero all Markbits. The Java language is relatively unusual in that it has no global variables. Thus the only sources of roots for garbage collection are stacks and constant/static class members. The next two steps of garbage collection are scanning all Stacks on the StackList and all Classes on the ClassList for Handles, and setting the corresponding Markbit (in time linear to maximum number 'static' variables in a Class multiplied by the maximum number of Classes).

The next step is a recursive sweep through the Handles, setting the Markbit of indirectly reachable Objects (in time linear to maximum Object complexity multiplied by the maximum number of Objects).

Unreachable Objects are then either queued for finalisation on the toBeFinalised queue or add()ed to the MemoryPool. Due to a finalisation timing restriction, the toBeFinalised queue must be sorted (in time linear to maximum object complexity).

If a compaction is necessary, it is performed using the two-pass method originating with (Morris, 1978), in which the first data word of the Object is stored in the pointer field of the Handle, and a pointer to the Handle in the first word of the Object (in time linear

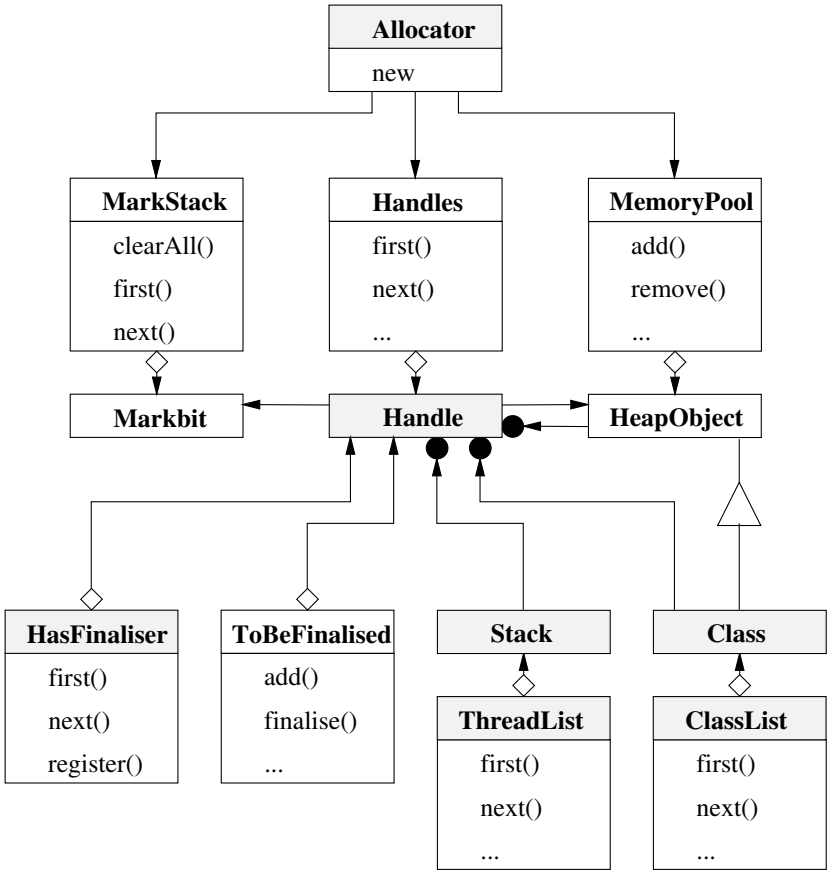


Figure 5.9: Memory management objects in Java

to the maximum number of objects)

Between each of the steps above, interrupts are checked for and if the collection is asynchronous, an interrupt causes collection to be aborted.

5.4.2 Object-Oriented Redesign

Figure 5.9 shows my object-oriented redesign of the Java collector.

5.4.3 Important Features

1. Finalisation is performed by a very low priority process which executes the finaliser and frees the object. The priority of the finalisation thread may be increased if memory is in short supply. Since the finalisers may be arbitrary code, no guarantees can be made on the threads performance, or whether it will ever finalise all objects.

   Finalisation in the Java language is performed in a rather odd manner, the draft standard (Sun, 1995) suggesting that finalisation not be relied on for program correctness. Objects are finalised in the order of instantiation, and the system make no guarantees that an object will be finalised before the objects it references. Requiring users of objects to track the order of creation would appear to be a breach of encapsulation of the same order as requiring them to track other users of the objects.

2. Java Classes are also garbage collected—they may be dynamically loaded and collected when no longer referenced. This is achieved by making the Class a sub-class of HeapObject.

3. Many of the data structures used by the garbage collector are also available to the user via Java runtime system. It's possible that applications which accessed these data structures in unusual ways could interfere with garbage collection.

Chapter 6

Design Patterns in Garbage Collection

> A garbage collector, in it's final shape, is a "low level" routine, which has to operate in an extremely complex system. In order to keep this complexity under control during each phase of the design process, the use of abstraction is indispensable. (Jonkers, 1983)

Design Patterns are a methods of capturing the design decisions and trade-offs within an object-oriented system in a reusable, standard format. While in theory they work well, doubts have been cast upon their effectiveness in the real world (Gabriel, 1996). This chapter aims to show several design patterns found in a predominantly mature, old implementations which emerged before design patterns became a force in software engineering. While this doesn't show that design patterns are necessarily useful in designing new systems, is provides evidence of "common ground" among software designers for design patterns to capture.

The design patterns (see Chapter 4) found in the garbage collectors examined fall into two groups: a) general patterns—those commonly found in both software and the literature on design patterns, such as those documented in (Gamma *et al.*, 1995), and b) specific patterns—those unique to a particular domain, in this case, garbage collection. As widely reported in the literature (Buschmann & Meunier, 1995; Gabriel, 1996), these two groups are a reflection of the fact that software faces both generic, domain-independent, problems found in a wide range of software systems and very specific, domain-dependent, problems which are dependent upon the application domain. This chapter aims to elaborate on patterns in of these groups, as they appear in garbage collection, and the garbage collectors examined in Chapter 5.

The general patterns found were the adapter, iterator and proxy patterns. The specific patterns were the RootSet and tricolour patterns, and their description here represents original work.

I shall use the 'Alexander' form to describe design patterns, a widely used method for describing patterns (Gamma *et al.*, 1995; Sane, 1995) outlined earlier (see Section 4.4). General patterns are described only in relation to their use within the domain of garbage collection and memory management.

The design patterns captured in the garbage collectors examined fall into two groups: (1) general patterns—those commonly found in both software and the literature on design patterns, such as those documented in (Gamma *et al.*, 1995), and (2) specific patterns—those unique to a particular domain, in this case, garbage collection. As widely reported in the literature (Buschmann *et al.*, 1996; Gamma *et al.*, 1993), these two groups are a reflection of the fact that software faces both generic, domain-independent, problems found in a wide range of software systems and very specific, domain-dependent, problems which are dependent upon the application domain.

The general patterns found were the adapter, facade, iterator and proxy patterns, each of which were put to specific uses in garbage collection. The specific patterns were the RootSet and tricolour patterns and their description here represents original work.

The following table summarises which of the patterns were found in which collectors.

Collector	Adapter	Facade	Iterator	Proxy	TriColour	RootSet
Baker78	no	no	yes	yes	yes	no
Boehm	yes	yes	yes	yes	yes	yes
Tolpin	no	no	yes	yes	no	yes
Java	no	yes	yes	yes	no	no

I will use the 'Alexander' form to describe design patterns, a widely used method for describing patterns outlined and used in (Gamma *et al.*, 1995) and (Sane, 1995).

6.1 Adapter and Facade Patterns

Adapters and facades are common in garbage collection, and in many situations the distinction between them is not clear. Functionally, adapters and facades each provide flexibility at the same point (the interface between the application and the garbage collector), but in different directions. Adapters allow subsystems to change their syntax independently, while facades allow subsystems to change their internal representation independently.

Schematically, the key difference between a facade and an adapter is that a facade provides a single integrated objects through which an entire subsystem or group of object may be accessed, while an adapter provides an altered interface to a single object to enable it to perform in a context which it was not designed for. Given the evolving nature of several of the garbage collectors examined (principally the Baker78 and Boehm collectors), and the large numbers of languages with similar but not identical memory management interfaces (C, C++, Pascal, Modula-2, etc), the distinction is necessarily not always clear.

6.1.0.1 Adapter

Adapters are common in memory management, for example, the Boehm collector, originally for C, uses an adapter to enable the same program to be used for C++. Most operating systems provide a single method of allocating heap memory, while languages such as C provide a plethora of library calls which allocate memory, each with slightly different semantics which makes adapters very useful. Languages such as C++, Oberon and Java have a 'new' operators, with divergent syntax, which allocate memory for an object. Using an adapter, a single garbage collector can be tailored to suit each language.

Name Adapter (also known as Wrapper).

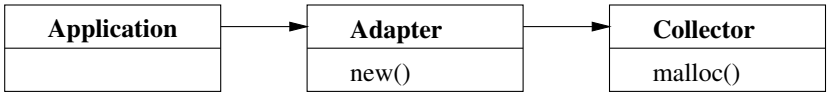
Intent Decouple the interface between the garbage collector and external system components.

Problem Different languages or language implementations require slightly different interfaces to the services provided by garbage collectors.

Solution Place an object between the collector and the external system to mediate their interactions.

Applicability Wherever the garbage collector is likely to be used with several versions of a system, or several systems, which are going to need slightly different interfaces.

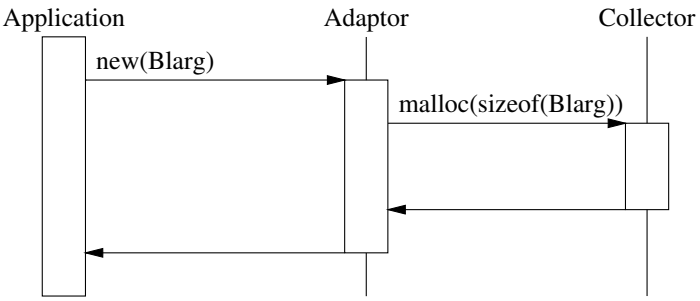
Structure



Participants

- Application
  - uses the services of Collector.
  - the syntax of the interactions with Collector may vary.
- Collector
  - provides memory management services to the Application.
  - may used with a variety of Applications, or a number of versions of the same Application each with subtly different interaction syntax.
- Adapter
  - mediates all interactions between a Collector and an Application.

Collaborations The following interaction diagram shows an Adaptor mediating between a C++ Application and a C Collector.



Consequence Because all interactions between the garbage collector and the external system are mediated by the adapter, the syntax of either garbage collector or system may change, with external change being limited to the adapter. This requires the overhead of a extra level of indirection, but when tuning for efficiency, macros or inlining may be used, allowing the overhead to be eliminated at the price of compile-time effort.

Implementation Commonly the Adapter is implemented as a wrapper around the garbage collector. When this occurs, inlining can result in the elimination of the overhead of having the Adapter object, while maintaining the full flexibility.

Sample Code The following C++ class is taken from the Boehm collector (Boehm *et al.*, 1991), where it adapts the C interface of the collector for use in C++. The GC class has a pair of methods, each a wrapper for a the appropriate C function, each method has explicit parameter and return types, so compile-time type errors are caught at the interface to the collector, not in it's internals.

```
class GC {
public:
  void* operator new( size_t size ){
    return GC_malloc( size );
  }

  void operator delete( void* obj ){
    GC_free( obj );
  }                                                                    10
}
```

6.1.0.2 Facade

Facades are used to capture the interface of a system (or sub-system), to simplify the exterior view of the system and hide internal changes of the system from its users. Further, facades also protect sub-system internals from unwanted examination and manipulation by users. This combination allows them to provide a package or sub-system form of encapsulation similar to that provided at the object level by most object-oriented programming languages.

In garbage collection facades are mainly used between the garbage collector sub-system and the application for encapsulation, namespace conservation and to accommodate the dropping of garbage collectors into systems designed to utilise traditional memory management systems.

Name Facade.

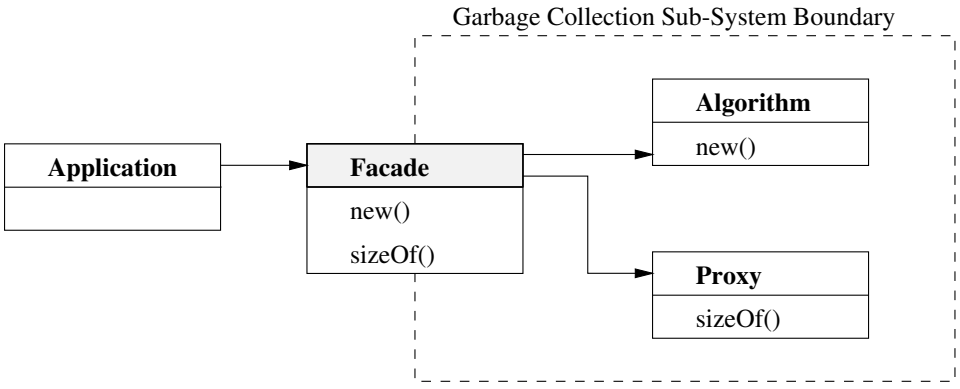
Intent Decouple interface between the garbage collector and external system components.

Problem External system components require a smooth, unchanging and clearly defined interface to a complex, evolving system.

Solution Decouple the specification of the interface from that of the garbage collector; completely hide the details of the language from the garbage collector, and *vise versa*.

Applicability Garbage collectors which are likely to change their internal structure or organisation during their lifetime, and external links to objects within this structure would impose unacceptable limitations of design freedom.

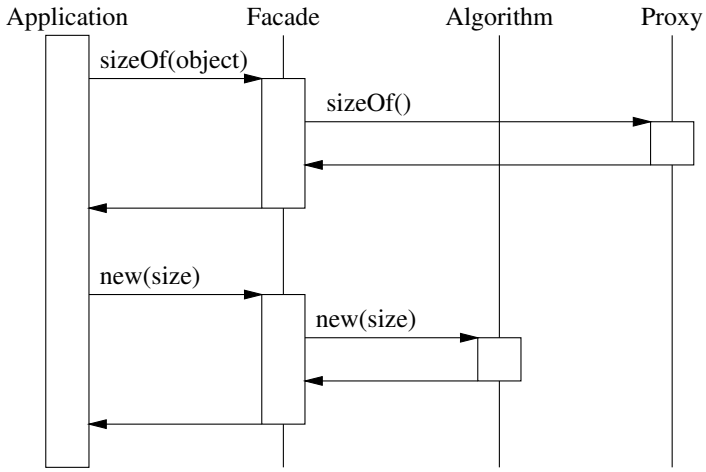
Structure



Participants

- Application
  - uses the services of garbage collector.
  - requires a clean, simple interface to the complex garbage collection sub-system.
  - interacts with the Facade as though the garbage collection sub-system were a single object.
- Facade
  - provides a clean, simple interface to the complex garbage collection sub-system.
  - is aware of some or all of the internal objects within the garbage collection sub-system.
  - directs method calls from Application to internal objects within the garbage collection sub-system.
- Algorithm and Proxy
  - Objects within the garbage collection sub-system.

Collaborations The following interaction diagram shows two method calls to a Facade being forwarded to other objects. The first method call, sizeOf() dispatched to the appropriate Proxy based on the object argument. The second method call, new() is always dispatched to the same object, Algorithm.



Consequence The interface to the garbage collector is narrowed to a single object. This requires the overhead of a extra level of indirection, but when tuning for efficiency, macros or inlining may be used allowing the overhead to be eliminated at the price of compile-time effort.

Implementation As suggested in (Gamma *et al.*, 1995), the application can be further decoupled from the collector by making the Facade an abstract class.

Sample Code The Boehm collector, while a large system, has a narrow facade to applications. The facade pattern is implemented in C, so the facade isn't an object but a header file, it still maintains namespace conservation—notice that each of the variables and functions begin with GC_. Comments are used to clarify the interfaces' use, so users of the collector need not look at the internals of the collector, only it's interface. The following file is an abbreviated gc.h from the Boehm collector (Boehm *et al.*, 1991), the facade between the collector and application.

```
#ifndef GC_H
#define GC_H
/* Public read-only variables */

/* Heap size in bytes. */
extern GC_word GC_heapsize;

/* Counter incremented per collection. Includes empty GCs at startup. */
extern GC_word GC_gc_no;
```

10

```
/* Using incremental/generational collection.*/
extern int GC_incremental;
```

```
/* Public R/W variables */

/* Disable statistics output.  Only matters if collector has been
 * compiled with statistics enabled.  This involves a performance cost,
 * and is thus not the default.
 */
extern int GC_quiet;                                                    20

/* Dont collect unless explicitly requested, e.g. beacuse it's not safe.*/
extern int GC_dont_gc;

/* Public procedures */

/* general purpose allocation routines, with roughly malloc calling convention */

  extern void * GC_malloc(size_t size_in_bytes);
                                                                        30
/* Explicitly deallocate an object.  Dangerous if used incorrectly.
 * Requires a pointer to the base of an object.
 */
  extern void GC_free(void * object_addr);

/* Explicitly trigger a collection. */
  void GC_gcollect();

#endif /* GC_H */
                                                                        40
```

6.2 Iterator

Iterators are objects which allow iteration over an aggregate object (a container) without exposing its internal structure. The primary action of all non-reference counting garbage collectors is performed through an iteration over the heap. Iterators are at the heart of garbage collection and a garbage collection cycle can be viewed as an iteration over each object in the heap. The iteration is recursive, with the recursive state being held in the TriColour rather than on the, each object directly or indirectly reachable from a root is marked reachable.

The robustness of iterators is also important. Robustness is the ability of software to withstand extreme conditions, especially those not anticipated by the designers of the

software. One critical aspect of robustness is resistance to interference—withstanding other operations on the aggregate during collection (Weide *et al.*, 1994; Kofler, 1993).

The traversal order (depth-first, breadth-first, hill-climbing or the more domain-specific hierarchical traversal) has important garbage collection implications in terms of locality of reference (Wilson *et al.*, 1991; Guggilla, 1994). For this reason, the exchange of iterators of differing traversal order is likely to be a key aspect in the optimising of garbage collectors.

There are three main types of iterations (and hence iterators) in garbage collection:

1. Iterations over roots—the set of pointers into the heap from outside. This iteration usually involves finding all pointers in global data structures and runtime stacks, and can be hard to incrementalise. Occurs once at the start of each garbage collection cycle.

2. Iterations over objects on the heap (sweeping)—this 'main' iterator is primed with the rootset during the flip() operation and its completion indicates the end of the garbage collection cycle. The tricolour holds the state of this iteration, and the read-barrier requires robustness from the iterator. Occurs once per garbage collection cycle.

3. Iterations over pointers within an object (scanning)—these are used to find which other objects a particular object references. Occurs once per reachable heap object per garbage collection cycle and unless special actions are taken, this results in the generation of vast numbers of single-use iterators. Fortunately, the scope of these temporary iterators is very limited and their lifetime very easily tracked (only one is in use at a given time), they can be reinitialised and reused. This eliminates all need to create new iterators during a collection cycle.

Name Iterator.

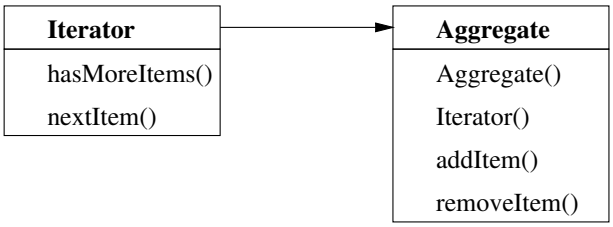
Intent Abstract iteration, iteration over roots, heap objects or references with an object.

Motivation There are many possible implementations of sets for implementing the heap data structures (linked lists, a mark stack, etc) each with differing advantages and disadvantages, in order to take advantage of these, standard interfaces are required, so one may be exchanged for another. The garbage collector must have a standard interface to these data structures, to allow it to 'walk,' or *traverse*, the structure, visiting each atom in turn.

Solution Detach the traversal order, mechanism and state from the rest of the system using an Iterator object which contains the state of the iteration, and through which all actions in the iteration are performed.

Applicability Wherever a data structure is likely to be traversed in multiple ways, or when the implementation of the data structure may change.

Structure



Participants

- Aggregate
  - A container or generator object.
  - Responsible for the creation, or retargeting (that is changing the container class it deals with), of Iterators.
- Iterator
  - Maintains a reference to Aggregate
  - Due to the vast number of iterations performed during garbage collection, may be retargeted and reused rather than destroyed and recreated.

Consequence The extra iterator object increases overhead, which may have some relative large methods making it hard to optimise, this is especially true of robust iterators.

Implementation Because of the vast number of iterations performed during the course of a garbage collection, the desirability of the garbage collector not using temporary objects (since it's likely to be invoked when there is little or no memory for their creation), reuse of Iterator object is desirable. By allowing them to be retargeted to iterate over another object, a single Iterator per client can be used.

Sample Code The Java runtime system maintains a ClassList of all Classes in the system, which must be swept at the start of each collection for Roots. ClassList has numerous

clients, each of whom are offered a wide interface. ClassList contains a Java Vector, with added type constraints and a slight narrowing of the interface. The last method, elements(), returns a ClassListIterator, which iterates over the ClassList.

```
import java.util.Vector;
import ClassListIterator;

class ClassList {

  Vector list;

  public          ClassList()                  { list = new java.util.Vector();     }
  public void     addClassStart(Class aClass)  { list.addElement(aClass);           }
  public Class    classAt(int n)               { return (Class) list.elementAt(n); }
  public boolean  contains(Class aClass)       { return list.contains(aClass);      }
  public Class    firstClass()                 { return (Class) list.firstElement(); }
  public int      indexOf(Class aClass)        { return list.indexOf(aClass);       }
  public boolean  isEmpty()                    { return list.isEmpty();             }
  public Class    lastClass()                  { return (Class) list.lastElement(); }
  public void     removeClass(Class aClass)    { list.removeElement(aClass);        }
  public int      size()                       { return list.size();                }
      // create a ClassListIterator wrapped around an Iterator
  public ClassListIterator elements() {
    ClassListIterator classListIterator = new ClassListIterator(list.elements());
    return classListIterator;
  }
}
```

ClassListIterator encapsulates an Enumeration. It should be noted that using Enumerations in this manner may not be suitable for all iterator implementations, since should the Vector be modified during iteration the common implementations of Java Enumerations (Gosling *et al.*, 1996) appear to have unspecified semantics. Enumeration is, however, defined using an interface, so where necessary it may be overridden to achieve the desired behaviour in incremental or multi-threaded environments.

```
import java.util.Enumeration;
import ClassList;

class ClassListIterator  {
  Enumeration enumer;
```

```
  /* construct a new ClassListIterator */
  public ClassListIterator(Enumeration enumer){
    this.enumer = enumer;
  }
  /* retarget this ClassListIterator */
  public void retarget(Enumeration enumer){
    this.enumer = enumer;
  }
  /* are there more Classes not yet seen ? */
  public boolean hasMoreElements(){
    return enumer.hasMoreElements();
  }
  /* return the next class */
  public Class nextElement(){
    return (Class) enumer.nextElement();
  }
}
```

6.3 Proxy

Proxies (Gamma *et al.*, 1995) are used in garbage collectors in three ways:

1. To control access to the object they guard. They are used to implement read- and write-barriers in the absence of (or as an alternative to) virtual memory.

2. To hide the movement of, the true location of, or changes in the content or location of, the object they guard. They may be used to conceal from the application movement of heap objects by the garbage collector.

3. To contain the per-object information about the state of the object they guard. They may be used in garbage collection to store "markbits" (the state of the tricolour) and the type of the object. Alternative implementations exist for each of these (bitmaps and tagless collection (Goldberg, 1991) respectively), but the information is commonly stored within a proxy.

In garbage collection, proxies are implemented in either of two ways. Firstly by storing the proxies separately from the object in a separate area of memory (such partitioning of memory can lead to maximum heap sizes being imposed, but can increase locality of reference in the collector, improving caching). Secondly by storing the proxy with the

object (which lends itself more readily to incrementalisation of proxy initialisation and resizing of the heap, but can have poor locality of reference). This second technique is used by traditional memory managers for languages such as C and C++, which commonly store a few bytes of data immediately before objects given to the application. (Wilson *et al.*, 1995)

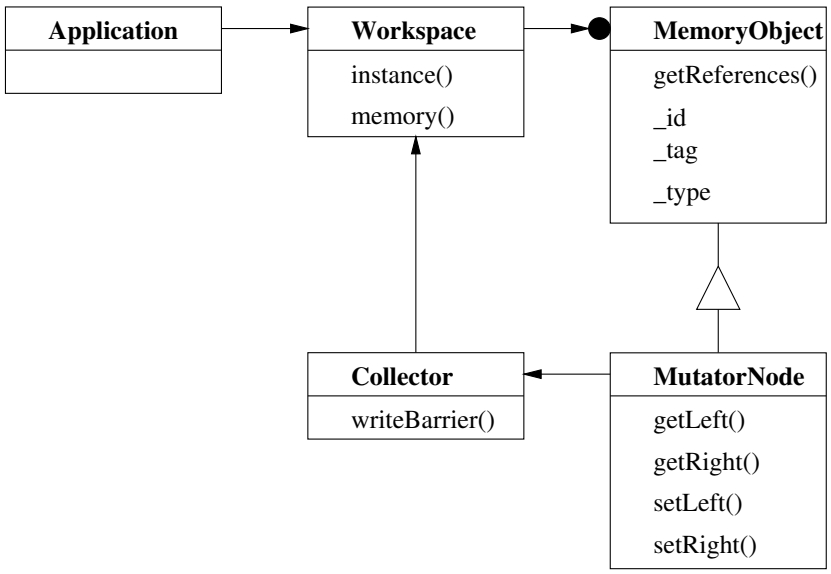
Name Proxy

Intent Provide a repository for per-object garbage collection state and functionality.

Motivation The garbage collector needs a) a place to store per-object data, b) an enforcement mechanism for read or write barriers, and, c) in a moving garbage collector, a mechanism to hide object motion.

Solution Use a proxy object for each application object. The application invokes methods of the proxy object as though it were the actual object, and the method invocation is passed on to the actual object (possibly after read or write barrier checks).

Structure The following is the structure of proxy pattern as found in my implementation of a garbage collector in Java:



Participants

- Workspace

  - maintains an array of MemoryObjects
  - maintains a list of free array entries
- MemoryObject
  - base class for all heap allocated objects
  - repository for per-object data (type data, markbits etc)
  - maintains an index _id into Workspace._memory indicating which element contains the reference to this object.
- MutatorNode
  - example user-defined heap object.
  - two data members, each accessed by accessors.
  - the set accessors contain calls to Collector.writeBarrier().
- Collector
  - represents the remainder of the collector
  - implements writeBarrier using (amongst other things) the fields in MutatorNode inherited from MemoryObject.

Applicability All incremental garbage collectors, and those non-incremental collectors which store per-object data with the object.

Consequence A level of indirection is added to method invocation. This is not necessarily a major problem in modern languages, which utilise inline method calls and perform similar optimisations.

Implementation Proxies have traditionally been implemented in memory management packages by placing an object of a few bytes between each normal heap object. These small objects are of a known size, and contain data about the following object, it's size, whether it's 'free' and if not, it's type. By subtracting a fixed number from a pointer to any heap object, a pointer to this data could be obtained. A more object-oriented approach to the problem is to have all heap objects derived from an AbstractProxy class, which contains this data.

Code Example The following example is lifted from my implementation of a garbage collector in Java. Workspace is a class representing the set of all application-visible heap objects. It holds all the objects in an array, for efficiency reasons, it exports references to this array. This array is similar to object tables in early versions of Smalltalk (LaLonde & Pugh, 1994)

```
public class Workspace {
  public static final short MAX_OBJECTS = 1000;

  static MemoryObject[] _memory   = null;
  private static Workspace _instance = null;

  public static Workspace instance() {
    if (_instance == null) _instance = new Workspace();
    return _instance;
  }                                                                    10
  private Workspace() {
    _memory = new MemoryObject[ MAX_OBJECTS ];
  }
  public MemoryObject[] memory() {
    return _memory;
  }
}
```

MemoryObject a the class representing a generic heap object, has a method for retrieval of references to other heap objects within the object, the location of this object within Workspace._memory, the type of the object and several mark bits. Java implements arrays as an array of references to objects, and each reference in the array is effectively a minimal proxy for the object it references. The application doesn't hold references directly the heap object, only indexes into the Workspace array.

```
class MemoryObject {
  short[] getReferences(){ return null; };
        short  _id;    // this objects place in the Workspace._memory
        byte   _tag;   // mark bits for the garbage collector
        byte   _type;  // the type of the object
      }
```

Objects written by the user, such as Node are transformed at compile-time to insert a write-Barrier check, and getReferences() overridden, as shown in MutatorNode.

```
public class Node {
  private short left = Short.MAX_VALUE;
  private short right = Short.MAX_VALUE;
```

```
  /** examine the value of the 'left' field */
  void setLeft(short s){ left = s;}

  /** examine the value of the 'right' field  */
  void setRight(short s){ right = s;}

  /** set the value of the 'left' field */
  short getLeft(){ return left;}

  /** set the value of the 'right' field */
  short getRight(){ return right;}
      }
```

```
final public class MutatorNode extends MemoryObject {

  /** the array of references in the node */
  private short[] references = {Short.MAX_VALUE, Short.MAX_VALUE};

  /** extract the references from the node, over riding the method
   * in MemoryObject */
  public final short[] getReferences(){return references;}

  /** examine the value of the 'left' field */
  public final short getLeft(){return references[0];}

  /** examine the value of the 'right' field  */
  public final short getRight(){return references[1];}

  /** set the value of the 'left' field, preserving the writeBarrier */
  public final void setLeft(short s){
    Collector.writeBarrier(this._id);
    references[0] = s;
  }

  /** set the value of the 'right' field, preserving the writeBarrier */
  public final void setRight(short s){
    Collector.writeBarrier(this._id);
    references[1] = s;
  }
      }
```

The Collector (representing the rest of the garbage collector) is the contains performs the write-Barrier check. The Collector can null, set or change the elements in Workspace._memory, during object freeing, object allocation and object moving respectively.

```
class Collector {

  static protected MemoryObject[] memory;

  public Collector() {
    memory = Workspace.instance().memory();
  }

  static void writeBarrier(short s){
    MemoryObject anotherObject = null;                                    10
    /* .... */
    memory[s] = anotherObject;
    /* .... */
  }
  /* .... */
}
```

6.4 TriColour

The TriColour marking is the theoretical proof of correctness on which incremental garbage collection rests (Dijkstra *et al.*, 1978). As such it typically features prominently in system descriptions and informal proofs, but it is not obvious from implementations, which are usually high-optimised for speed (Boehm & Weiser, 1988).

The TriColour is also the repository for the state of the garbage collectors traversal of the heap. All incremental garbage collectors which have been studied in this work incorporate tri-colour marking or an equivalent data structure. Non-sweeping (pure reference counting) collectors incorporate neither TriColour nor an equivalent data structure.

Name TriColour.

Intent Maintain the tri-colour proof-of-correctness.

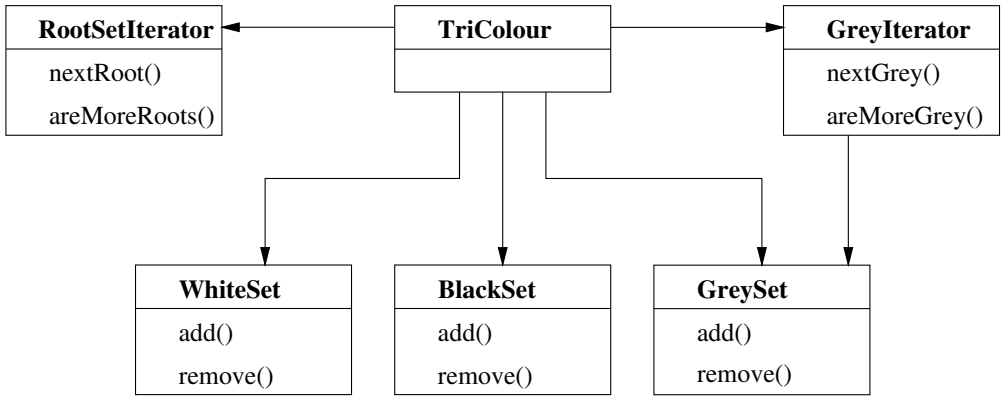
Motivation the tri-colour proof-of-correctness is the theoretical basis for incremental collection, but is commonly obscured by the need for 'speed' and efficiency, leading

to difficulties in ensuring algorithmic correctness in the face of application mutation of the heap.

Solution Clearly isolate the elements of the tri-colour marking proof as an abstract data type, removing the proof of correctness from the implementation of the collection.

Applicability Incremental garbage collectors using the tri-colour proof-of-correctness.

Structure



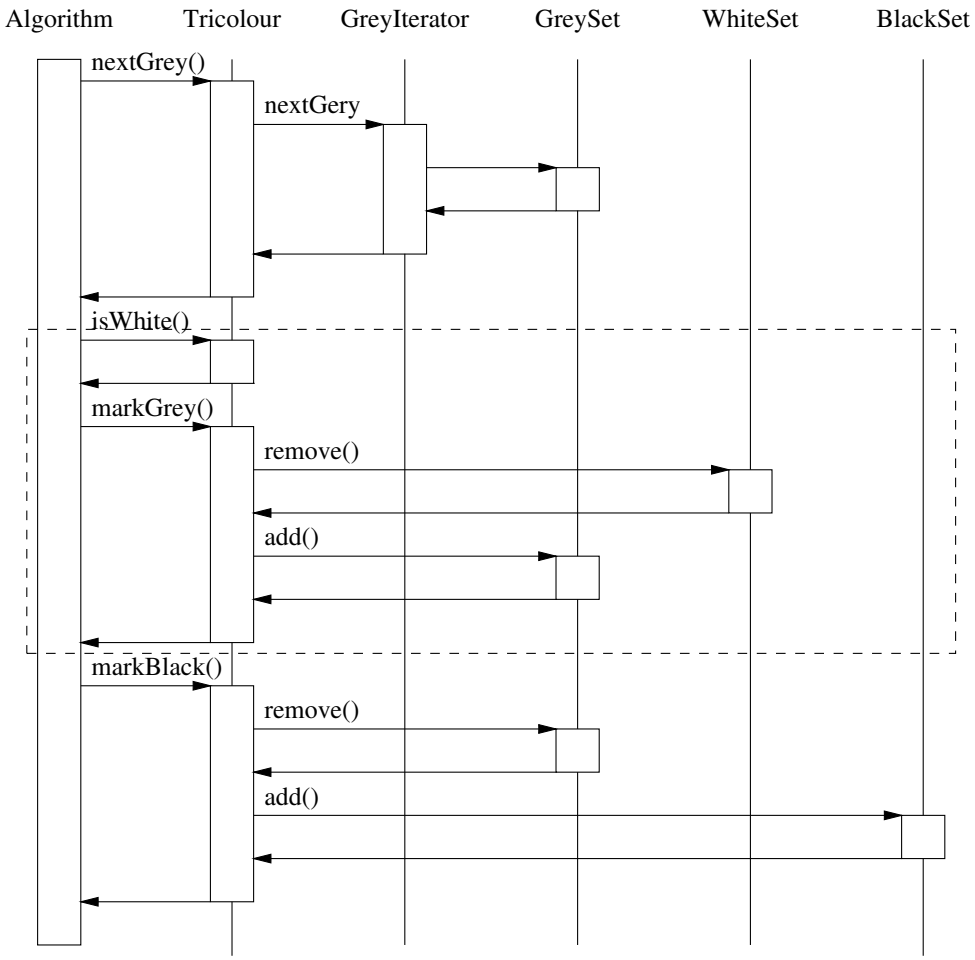
Participants

- TriColour
  - implements the tri-colour proof-of-correctness
- RootSetIterator
  - the source of roots of the iteration over the heap
- BlackSet
  - keeps track of those heap objects which are known to be reachable and have been examined for other objects.
- GreySet
  - keeps track of those heap objects which are known to be reachable but have not yet been examined for other objects.
- GreyIterator
  - an Iterator through GreySet

- determines whether the garbage collection is a depth- or breadth-first iteration through the heap

- WhiteSet
  - keeps track of those heap objects with unknown reachability

Collaborations The following diagram shows an Algorithm (representing the rest of the garbage collector) performing a tracing a single object. It first gets the object to trace with a call of nextGrey, which is forwarded to GreyIterator, which in turn gets it from the GreySet. The Algorithm then traces the objects referenced by the grey object: Algorithm calls IsWhite() on the first, to determine whether it has already been found to be reachable. Finding that it hasn't, it calls markGrey to remove() it from WhiteSet and add() it to GreySet. This step (dotted outlined) is repeated for each reference in the original grey object. The Algorithm then calls markBlack to indicate that this original object has been finished with and should be remove()ed from the GreySet and add()ed to the BlackSet.



Consequence In all of the garbage collectors examined, the TriColour featured far more prominently in the system description than the implementation. By embedding the theoretical proof from the implementation, a wider choice of implementation options is available while transparently preserving the necessary conditions for tri-colour marking. Because the implementation is closer to the theoretical proof, the chance of small, race condition-like, mistakes creeping into the collector is reduced.

Implementation Past implementations have not implemented the tricolour explicitly, but described it in their documentation, and presented code which contained the tricolour only implicitly. Membership of each of the sets is usually indicated using a per-object bit pattern stored in each object, enabling constant time membership tests.

Code Example The following example is from my implementation of a garbage collector in Java. It extends the basic tri-colour as outlined by (Baker, 1978) by also including management of unreachable, but unreclaimed objects. This is to allow the objects to be reclaimed (and if necessary, finalised) incrementally, rather than during the flip, as Baker does. Unlike the scheme described in (Baker, 1995a), free objects are managed externally, *unreachable* objects are those known to be unreachable by the application differing from Baker's *ecru* objects in that they are passed. register() is used to add an object, and deRegister() called when the object has been reclaimed.

```
import java.util.Enumeration;
/**
 * Interface TriColour, the interface between the garbage collection
 * algorithm and the garbage collection state.
 *
 * White objects are of unknown reachability.
 *
 * Grey objects are reachable, but have pointers which have not been
 *   examined. they represent the current fringe of the traversal of
 *   the collector through the heap graph.                          10
 *
 * Black objects are reachable, and finished with.
 *
 * @version  0.2, 12 May 1997
 * @author   stuart yeates
 */

public interface TriColour {
```

```
/**
 * is this object in this TriColour ?
 * @return true if object is grey
 * @param object  the object under consideration
 */
boolean isMember(Object object);
/**
 * is this object marked grey ?
 * @return true if object is grey
 * @param object  the object under consideration
 */
boolean isGrey(Object object);
/**
 * is this object marked white ?
 * @return true if object is white
 * @param object  the object under consideration
 */
boolean isWhite(Object object);
/**
 * is this object marked black ?
 * @return true if object is black
 * @param object  the object under consideration
 */
boolean isBlack(Object object);
/**
 * mark an object grey
 * @param object the object to be marked
 */
void markGrey(Object object);
/**
 * mark an object white
 * @param object  the object to be marked
 */
void markWhite(Object object);
/**
 * mark an object black
 * @param object  the object to be marked
 */
void markBlack(Object object);
/**
 * are there more grey objects ?
 * @return true if there are more grey objects
 */
```

```
  boolean areMoreGrey();
  /**
   * which is the next grey item ?
   * @return the next grey Object
   */
  Object nextGrey();
  /**
   * start a new garbage collection cycle
   */                                                              70
  void flip(Enumeration rootset);
  /**
   * register a new Object
   */
  void register(Object object);
  /**
   * de-register a Object. this should only be called by the finaliser,
   * after it is certain the application (including finalisers etc) has
   * completely finished with the object.
   */                                                              80
  void deRegister(Object object);
}
```

6.5 RootSet

Roots are the starting points for the collector's iteration over the heap. They are held in a wide variety of locations including the stack, the global data area, across the network (in a distributed system), other heaps (in a multi-heap system), and in persistent object stores. Roots may be updatable (writable), and have varying costs of updating. Roots may time-out or otherwise become stale.

To deal effectively this complexity, a common interface is needed.

Name RootSet.

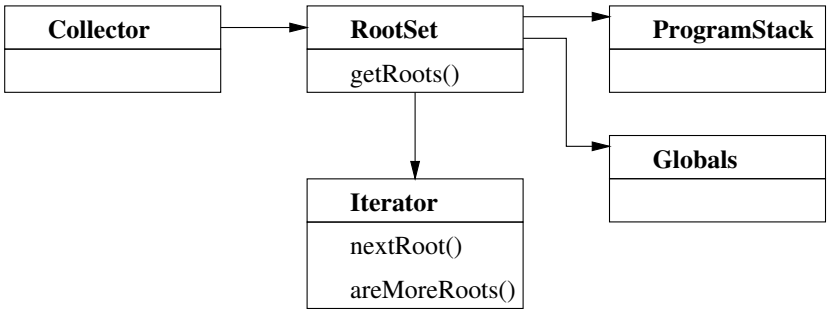
Intent Abstract the generation and retargeting of RootSets.

Motivation In some cases the roots are read directly from the execution stack and globals (for example the Boehm collector), while in other cases the roots are derived from a separately maintained data structure which within the garbage collector (for example the Tolpin collector).

Solution Create an abstract interface, through which all roots may be interfaced, along with a container of current roots which may be iterated through at the start of each garbage collection.

Applicability All sweeping complex garbage collectors use sets of roots (non-complex collectors include single rooted, single generation, lisp collectors), and many (for example generational or thread-aware collectors) have multiple sources of roots. If these sources are significantly different, some from of abstraction is necessary.

Structure

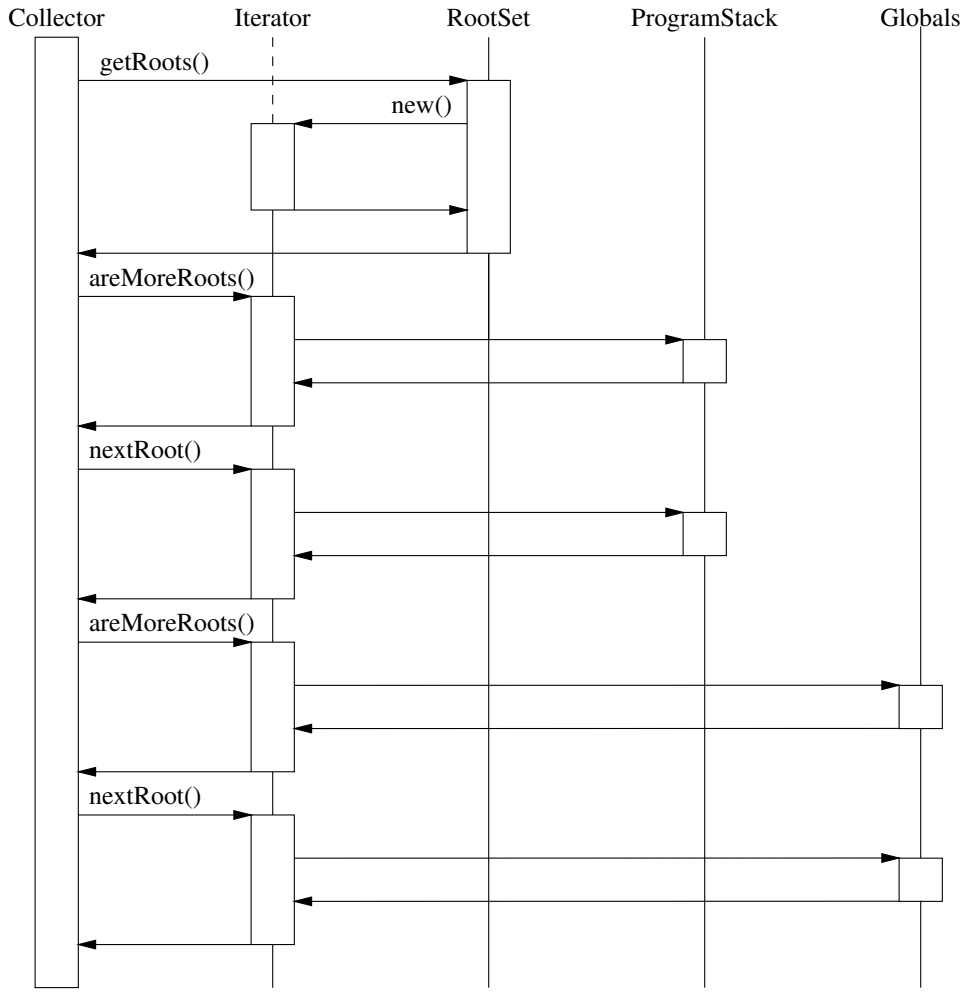


Participants

- Collector
  - needs to iterate over all the roots in a system regardless of their source.
- RootSet
  - responsible for generating, or retargeting, Iterators.
  - may be a container object or an interface to another sub-system, such as the run-time stack.
  - if roots are generated from more than one source (in this case ProgramStack and Globals, responsible for the creation of a single iterator which iterates over both sources.
- Iterator.
  - responsible for managing the Collectors iteration through the RootSet.
- ProgramStack
  - a source of roots

- Globals
  - a source of roots

Collaborations The following diagram shows a Collector iterating over a pair of sources of roots ProgramStack and Globals.



Consequence Separates the traversal of the roots from the identification and maintenance of the roots. By allowing roots to be abstracted independently of their source they clarify the tricolour and enable generations within generational collectors to be decoupled from each other.

Implementation RootSet is effectively just a standard interface to a data structure and associated iterator. Unlike iterator the iterator pattern, however, arbitrary operations may be needed on the abstract data type before the RootIterator can be

used. In the general case, these operations are used to enforce both availability and temporal constraints on the roots.

Undoubtedly, the most common source of roots is the program execution stack. There are three methods of implementing the interface between the program execution stack and the garbage collector, each using a different level of granularity:

1. per frame—whenever a stack frame is pushed or popped, the pointers on the heap from that frame are added to the RootSet. The cost of this method is extremely high, but proportional to the number of method invocations, but evenly distributed across all invocations, making maximum latency low. Iterating over the RootSet is rapid, and may be performed without reference to non-garbage collection structure. The Tolpin collector manages the RootSet in this manner.
2. per stack—whenever a new stack is created, a pointer to the base of the stack is added to the RootSet. When the roots are iterated over, each stack frame must be examined, and the collection of roots within iterated over. The cost of this method is much lower, but entire incurred during the flip() operation, increasing the maximum latency. Because stacks are associated with threads, which may have priority associated with them, this method of RootSet implementation may be adapted to iterate over high-priority threads first, enabling high priority threads, which typically have small stacks, to be iterated over first. The Boehm collector uses this method.
3. per program—a single root is used. This is the trivial RootSet of those implementations which create the stack on the heap. The Baker78 implementation uses this method, but takes care that heap objects representing the program execution stack get scanned early in the collection.

Sample Code The following example is from my implementation of a garbage collector in Java.

```
package openKernel.objectManager;
/**
 * Interface RootSet, the interface between the RootSet and the garbage
 * collector.
 *
 * Note: that RootSet CANNOT be implemented using a Snaplist, as the fields in
 * MemoryObject used by Snapist are used by the Snaplists in TriColour.
 *
```

```
 * Note: the iterated over are not necessarily the roots added and removed
 * from the set. For example, an implementation could add and remove stack
 * bases, but return an iterator over each frame in each stack.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

public interface RootSet extends ObjectSet {

  /**
   * Add a root to the RootSet
   * @param root the root to be added
   */
  void addRoot(MemoryObject root);

  /**
   * Remove a root from the rootSet
   * @param root the root to be removed
   */
  void removeRoot(MemoryObject root);

  /**
   * Returns an Iterator over the container.
   */
  Iterator  newIterator();

      };
```

Chapter 7

A Garbage Collector designed using Design Patterns

> The decomposition of a compacting garbage collector into a garbage collector and a compacter is only a "conceptual decomposition". It is analogous to the decomposition of a compiler into "phases": lexical analysis, parsing, semantical analysis, code generation, etc.. By merging these phases (such as in a one pass compiler) a considerable increase in efficiency con be obtained. (Jonkers, 1983)

A garbage collector was designed and implemented with the design patterns isolated in chapter 6. The design and implementation had several goals, including testing the design patterns and the production of a garbage collector suitable for use with **OpenKernel** (de Champlain, 1995). The **OpenKernel** kernel is a micro-kernel designed to be portable (de Champlain, 1996a; de Champlain, 1996b) and object-based, written in the language Java (Gosling *et al.*, 1996), and compiled for the Java Virtual Machine (Lindholm & Yellin, 1996). The development platform used was various versions of the Java Development Kit (JDK1.0.2, 1996; JDK1.1.1, 1997), the languages reference implementation, running on Solaris 2.5.

7.1 Requirements Analysis

Because the garbage collector was being designed separately from the rest of the system, a separate requirements analysis was performed.

The garbage collector is responsible for:

- Detecting objects no longer in use.

- Notifying the memory manager of finalised objects for reuse.
- Guaranteeing that memory will not be exhausted, when memory usage stays within certain precalculated bounds.
- Performing all the above duties in tightly bounded time, except for initialisation.
- Performing all the above duties without the creation of temporary, internal objects, except for initialisation.
- Guaranteeing that all unreachable objects will be reclaimed in bounded time.

The garbage collector is *not* responsible for:

- Management of unused memory.
- Placement or allocation of new objects.
- Defragmentation of the memory pool.
- Management of run-time type information.
- Invocation of finalisers—user level finalisation is not present in the target language.
- Synchronisation control—the runtime systems shall ensure that only one thread is active in garbage collector code at any one point.

The garbage collector requires the following services from other system components:

- The garbage collector needs a method of finding pointers into the heap (roots). This may be done in one of several ways:
  * If the execution stacks are allocated in garbage collected memory, the garbage collector only need to know about the thread creation, the collector can use normal tracing techniques on them.
  * If the execution stacks are in non-garbage collected memory, the garbage collector needs access to an iterator over them (this may be done on a per-thread basis).

  If non-stack pointers into the heap exist, the garbage collector will need access to an iterator over them.
- The garbage collector is notified of the creation of every heap object. Objects such as thread stacks need not be registered, if their reclamation is not under

the control of the garbage collector, and any pointers into the heap they contain are made available as roots.

- Because the garbage collector is to be accurate, it must be able to locate references within objects, which is usually achieved via a type-system.
- If the collector needs to be 'tuned' for real-time performance, to maximum time for a register(), the following information must be available, their main use is in balancing the size of gcQuanta(). This can be thought of as a scheduling problem: how to minimise the maximum execution cost of gcQuanta(), given the constraint that garbage collection must be completed before memory exhaustion, and the desirable property of completing garbage collection in the minimum time. This information is required at compile-time.:
  * The maximum object complexity: the number of pointers to other objects in the most pointerful type.
  * The maximum execution time of the finaliser with largest maximum execution time.
  * The maximum execution time of the type interface methods used.
  * The maximum number of heap objects.
  * Type/size information on objects.

7.2 Design

Figures 7.1 and 7.2 shows the initial class decomposition diagram for the garbage collector.

- Collector—the interface (see Section 6.1) to the garbage collection sub-system. All interactions between the garbage collector and other sub-systems (other than the type-interface) are via the Collector. Responsible for the interface between the garbage collection and other subsystems.
- Algorithm—the primary location of the garbage collection logic. Responsible for maintenance of performance guarantees, initiating flip()'s and definition the read- or write-Barrier as necessary.
- TriColour—a class representing a TriColour, the primary location of garbage collection state (see Section 6.4). There are as many TriColours in a garbage collector as

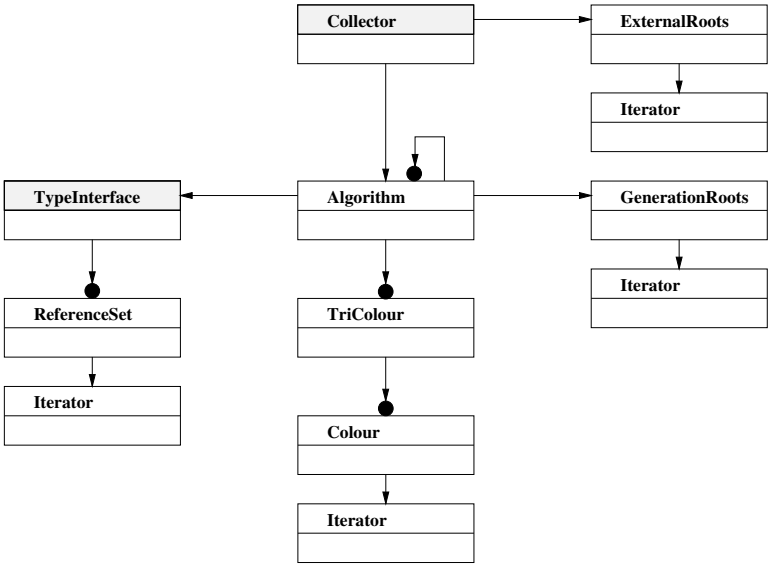


Figure 7.1: The initial class decomposition diagram of the objects in the garbage collector.

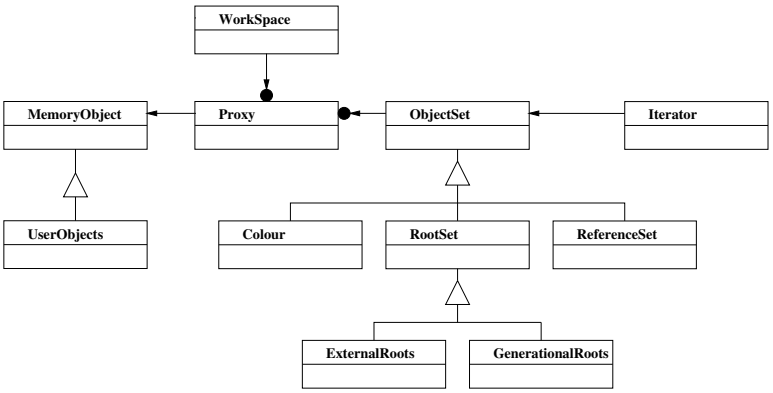


Figure 7.2: Class decomposition diagram showing ObjectSet and related classes. Note that due to the simple generational model adopted, GenerationalRoots aren't used in the implementation.

there are generations (one for non-generational systems). Primary repository for collector state. Synchronisation guarantees (for manipulating if internal state) provided by external system. A fourth colour was added to the TriColour, called Unreachable, which contains those objects which were in the TriColour but are no longer reachable. This is to enable these objects to be returned to the control of the memory manager incrementally, rather than all at once during the flip(). Valid operations are register(), deRegister(), isMember(), isGrey(), isWhite(), isBlack(), markGrey(), markWhite(), markBlack(), areMoreGrey(), nextGrey(), flip(), areMoreUnreach() and MemoryObject nextUnreach().

- TypeInterface—the interface between the Collector and the type system. Responsible for all type-interactions.

- ObjectSet—aggregate objects (objects formed from many other objects). Each aggregate has a different set of operations and, potentially, a different implementation (see Figure 7.2). The aggregates are

  - Colour—an aggregate of all the objects of a single colour in a TriColour. Valid operations are isNotEmpty(), isEmpty(), newIterator(), remove(object), insert(object) and next().
  - ReferenceSet—an aggregate of references within a heap object. A Java native array. Valid operations are newIterator().
  - RootSet—an aggregate of objects to which are 'roots'.
    * ExternalRoots—an aggregate of objects to which references exist from outside the garbage collector, for example the stack or global variables.
    * GenerationalRoots—an aggregate of objects to which references exist from older generations.

The following classes from **OpenKernel** were used, but are not, strictly speaking, part of the garbage collector. The entire memory manager is very flexible, for example the Workspace–MemoryObject interaction allows moving or coping collectors to be implemented, a dimension of flexibility not utilised in my implementation.

- Iterator—an iterator over an ObjectSet.

- MemoryObject—the class representing all user objects. All access to MemoryObjects is via a Proxy, a pointer to the stored in the Workspace.

- Workspace—a singleton object responsible for allocating MemoryObjects, contains an an array of references to all application-visible MemoryObject. This relationship is elaborated on in the example given in section 6.3.

However, after an examination of the properties of the various data structures (see section 7.3.2, it became increasingly clear that the overhead of inter-generational roots was likely to become high. For this reason, a very simple model generational model was used, in which all objects which survive a flip() are promoted. When working in a incremental manner, all objects which have references to them written to heap objects are guaranteed to survive a flip, but objects which are purely referenced from non-heap areas (for example the stack) are not. This allows temporary objects whose scope is limited to a stack frame to be allocated and collected very cheaply.

The Iterator over Colour needed to be *robust*, in the sense of (Kofler, 1993), in that arbitrary objects could be added to, or removed from the Colour while the iteration was in progress. While such robust iterators are possible, and known, they are uncommon, have considerable overhead and are generally not widely used (Murray, 1993). The proof, however, only calls for a iterator over a colour at any time, the grey Colour being used as a stack or a queue (depending on whether the heap traversal was a depth- or breadth-first one). Combining Colour and it's Iterator into one object saves two objects per TriColour, and several object interactions. This limitation of one Iterator per Colour is similar to Eiffels cursor (Meyer, 1994).

Figure 7.3 shows Figure 7.1 updated for these implementation decisions.

7.3 Implementational and Performance Issues

7.3.1 Choice of Algorithms

Two major criteria affected my choice of garbage collection algorithm: firstly the suitability for embedded/real-time systems and secondly ability to experiment with as many variants as possible using the least implementation effect.

Figure 7.4 shows the main garbage collection algorithms. The four scanning algorithms (Mark-and-Sweep, Mark-and-Compact, Semi-Space Copying and Treadmill) may be incrementalised or generationalised (or both). Some minor hybrid algorithms are omitted for clarity.

Mark-and-Compact algorithms are unsuitable for real-time applications, because efficient heap compaction (Morris, 1978) cannot be incrementalised. Non-Treadmill Semi-Space Copying algorithms (Wilson, 1992) would both require a read barrier and handle

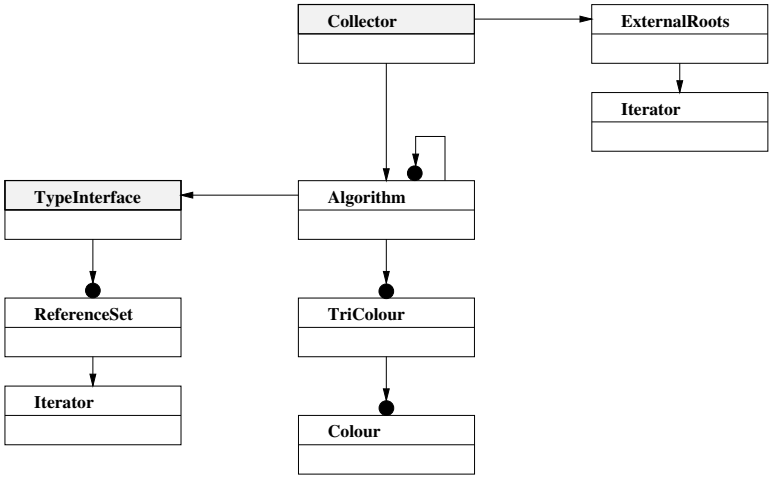


Figure 7.3: The final class decomposition diagram of the objects in the garbage collector, after the generational mechanism has been removed, and Colour and it's Iterator have been united.

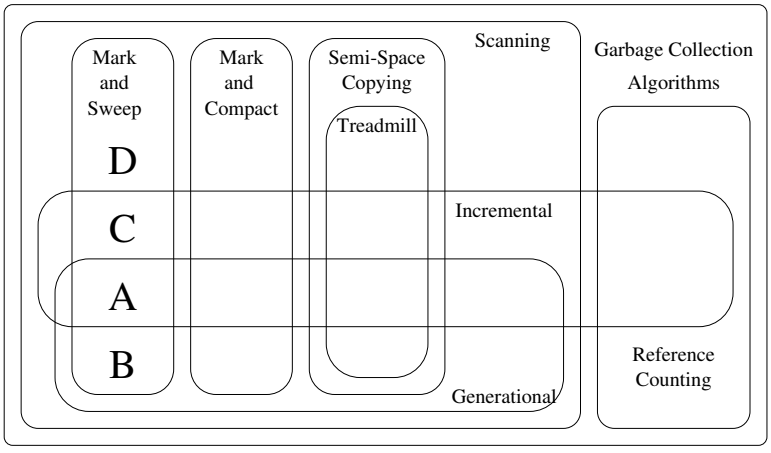


Figure 7.4: A Venn diagram showing the relationships between the main garbage collection algorithms

pinned memory—memory accessed by system components unable to use proxies, for example direct memory access hardware—only with considerable added complexity (Yip, 1991). The Treadmill Semi-Space algorithm (Baker, 1992) requires that free memory be handled within the collector (Wilson, 1992). Reference counting algorithms are sufficiently different to scanning algorithms that implementing them both, within the same framework, would add considerable design and implementation effort.

Incremental Mark-and-Sweep times it's flip() on heuristics and hence is usually unable to provide guarantee that garbage collection will finish before memory is exhausted. However, by setting the collection rate sufficiently high[1], this guarantee can be provided. If implemented in it's incremental, generational form **A**, it is tunable in two dimensions, the increment size and the number of generations. The lower bound on increment size is that needed to guarantee finishing the collection before memory exhaustion, the upper bound (an infinitely large increment) produces a non-incremental generational collector **B**. The number of generations may be varied from one (producing a non-generational collector **C**), with no upper bound. If there is a single generation and an infinite increment size **D** is produced.

From this I conclude that the best algorithm to implement is a generational mark and sweep garbage collector.

7.3.2 Choice of Data Structures

Lists, sorted trees of various types and bitmaps are the data structures traditionally used in memory management and garbage collection systems (Wilson, 1992), but most such systems, the average or amortised cost of operation is important, rather than the worst-case time. Thus while some systems can trade (for example) a slightly slower join for a faster membership test, this cannot be done in a real-time system, where the sort and/or hard deadlines must be met.

Ideally, a real-time garbage collector should be able to perform all operations in constant time, independent of the number of objects on the heap, the number of pointers in them, the number of pointers in them and the number of pointers in them. To achieve this, the data structures within the collector must provide operations which are are independent of these. Three main data structures are present within a garbage collector: the aggregate or container of objects of each colour within the TriColour; the RootSet between the collector and the rest of the system and the RootSets between generations within the

[1] generally tracing two objects per object allocation, but always calculable and calculated at, or before, compile time

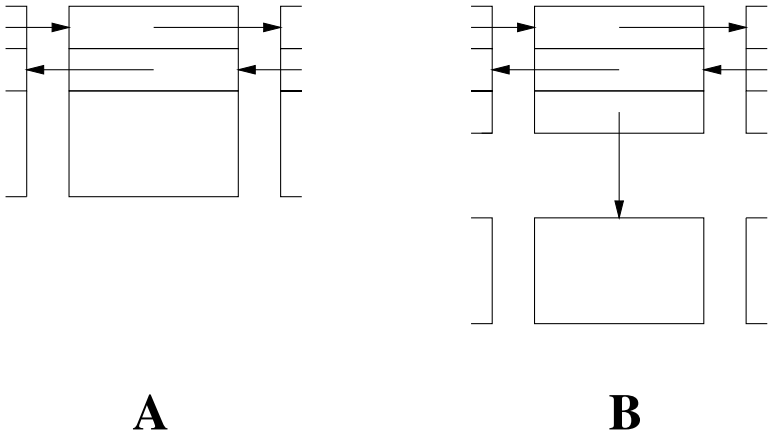


Figure 7.5: Interior (A) and exterior (b) doubly linked lists.

collector (if any).

Linked lists and sorted trees (heaps, b-trees etc), arrays (or tables) of pointers and arrays (or tables) of counts were all considered for use in the garbage collector. Hash tables were not considered, while they can be 'tuned' if much is known beforehand about the data (or if time can be taken on-the-fly to rehash them), they do not, in general, perform in real-time.

Doubly linked lists (internal can have objects insert or removed in O(1), and because only the ends need be dealt with, joins can also be performed in O(1). Internal lists need two pointers in each object in the data structure, one to the preceding object, and one to the next.

Doubly linked lists (external) can have objects insert O(1). Because there is no pointer from the object to the node in the structure containing the reference to this object, a linear search of the list is required to find the node, making removal O(n). Adding a pointer from the object to the node reduces this to O(1), but removes the possibility of multiple membership. Joins can be performed in O(1), for the same reasons as an internal list. External lists need the same next and previous pointers as internal lists, as well as a pointer to the object, this is shown in figure 7.5.

Sorted trees (internal) are sorted by object location in memory (to allow merging of unused objects etc), which is only loosely related object connectivity. Inserting and removing objects is O($\log n$). When two trees are joined (merged), in the worst case,

every object must be examined, making it O(n). Internal sorted trees require three pointers (left, right and up) for each object.

Sorted trees (external) are also sorted by object location in memory. Inserting and removing objects is O($\log n$). When two trees are joined (merged), in the worst case, every object must be examined, making it O(n). External sorted trees require three pointers (left, right and up) for each object, in addition to a pointer to the object.

Arrays of pointers can be used as a queue (either FIFO or FILO) for the storage of pointers to objects, they work very effectively is this mode of operation matches usage of the data structure, but this can break down if perturbed. If, for example, the data structure is to be generated, iterated through a single time, and then regenerated, arrays of pointers are ideal. They are also the data structure of choice for storing references to other objects within an object, but this usage requires only iteration and dereferencing operations, not those discussed here. The following figures assume the array is being used as a pure FIFO or FILO queue using pointers to the first and last elements in the queue. Insertion and deletion may be performed in O(1), but joining requires copying one of the arrays into the other, in time proportional to the size of the array. The array must be of a fixed length for the duration of it's lifetime, so the space overhead is proportional to the maximum number of objects, not the current number of objects (n not m).

Arrays of counters assume that there's an explicit counting of objects. The simplest array of counters is a bitmap, but this can be extended to allow an object to be in the structure more than once. In the past bitmaps have been used because bits could be packed efficiently (in both speed and cpu cycles) into bytes, this is not the case in Java, which has no bit or boolean type. Insertion and detection take O(1), but join requires examination of every entry, making it proportional to the maximum number of objects, O(n).

All of these were considered in combination with a short bit-pattern for structure-membership. Using only a single byte per object to store the identity of the structure it belong to, it is possible to check an object for membership in a set with a single integer. These data structures are shown in Table 7.1.

The number of such data structures within a generation collector is also important. An approximation:

Data Structure	membership test	multiple membership	insert	remove	join	overhead (bytes)
doubly linked list (int.)	O(1)	no	O(1)	O(1)	O(1)	$(2B)m$
doubly linked list (ext.)	O(1)	yes	O(1)	O(n)	O(n)	$(3B)n$
sorted tree (int.)	O(1)	no	O($\log n$)	O($\log n$)	O(n)	$3Bm$
sorted tree (ext.)	O(1)	yes	O($\log n$)	O($\log n$)	O(n)	$4Bn$
array of pointers (ext.)	O(1)	yes	O(1)	O(1)	O(m)	Bn
array of counters (ext.)	O(1)	yes	O(1)	O(1)	O(m)	Cm

Table 7.1: Data structure properties. For each data structure, the cost of checking membership of a single item, whether or nor an item can be a member of the same structure multiple times, the cost or inserting or deleting a specific item, the cost of joining two such data structures, and the memory overhead required by the data structure. Where n is the maximum number of items in the data structure, m is the maximum number of heap objects, p is the maximum number of pointers in a heap object, B is the size of a pointer to an object or an index to a proxy (1–4 bytes) and C is the size of the counter used. All objects are assumed to have an overhead of A, the size of a membership flag (typically 1 byte). 'int.' (internal) indicates that the memory overhead is distributed as fields in the items, 'ext.' (external) indicates that the memory overhead is external to the items.

- One RootSet of pointers into the heap from elsewhere, internal data structures, which don't allow multiple membership, are not suitable for this, since there may be an unbounded number of roots pointing to a single object.
- Three per TriColour, to hold the colours, either internal or external internal data structures are suitable for this, as a single object may only be a single colour in a single Tricolour at any point.
- In a Generational system, each TriColour requires a one data structure per older generation, to hold the roots pointing from the older generation to the younger. The oldest generation needs no structures, the i generation needs a structure from each older generation, or $i-1$ structures, thus for n generations, $(1/2n^2) - n$ data structures are required.
- The data structure within each object which holds the references to other objects is not considered here, because it is not, strictly speaking, part of the garbage collector. There appears to be universal use of arrays for this purpose.

This count is approximate, since some of these may be eliminated by tuning or implementation—for example a youngest generation which prompts all objects every flip() requires no RootSets, since it when the RootSet is applied the generation is empty, thus there can't be any roots in it. Alternatively, others may be introduced—for example an unreachable set in each generation to all object de-allocation or finalisation to be amortised across all object allocations.

The design uses internal linked lists for Colours, and either external linked lists or an array of counts for the ExternalRoots. No generational data structures are used. Arrays of pointers will be used for references within objects.

ExternalRoots represent a problem, in that, for a procedural language, there are a vast number of insert(object) and remove(object) calls (one for each time a reference into the heap is written to the stack). If this overhead is unacceptably high, ExternalRoots can be implemented as a place holder which merely generates an iterator over current references into the heap from the stack. This Iterator could parse the stack, thus eliminating the cost of the writes to the stack, but possibly increasing the work to be done during the flip(), and hence the maximum time to register an object.

7.3.3 The Write-Barrier

A barrier is a check on each read or write of an object to/from the heap or stack to determine which of the tricolour sets the object is in. The choice of a read or write barrier is closely linked to the choice of garbage collection algorithm: all incremental algorithms require a barrier and all defragmenting incremental algorithms (those without proxies) require a read barrier. A write barrier requires that all writes to the heap of pointers to the heap be checked. A read barrier requires that all reads of pointers to the heap from the heap be checked. Both a read and read barrier have the same potential complexity, but due to the fact that there are far fewer writes than reads, a write barrier is far more efficient.

Because mark-and-sweep is a non-copying collector, a write-barrier is sufficient, rather than the less efficient read-barrier.

7.3.4 Finding Pointers

In is necessary to find, and iterate over, pointers in heap objects. An interface to the type system was provided. The current type system, however, is design for compile-time not run-time access. For this reason it may be necessary to establish a cache of mappings between types and the arrays of references within objects of those types.

7.3.5 Freeing of Objects

Free heap objects are to be managed externally to the garbage collector, in the Workspace. The return of unused (unreachable) objects to the Workspace requires one put() call per object to be freed, meaning that if all unreachable objects were returned at once, this

operation would be proportional to the number of unreachable objects and hence the number of objects on the heap, since all of the heap may become unreachable. For this reason, objects *must* be freed incrementally.

Returning one unreachable object per object allocation would be sufficient, but may lead to excessive fragmentation of the externally managed free objects. Studies have shown (Wilson *et al.*, 1995) that memory managers defragment best when they have many objects in them, increasing the probability that two of the objects are adjacent and may be merged. Returning one unreachable object per object allocation minimises the memory in the memory manager, while returning all at once maximises it. I conclude that returning a small number (two or three) unreachable objects per object allocation will be sufficient. If this number proves unacceptably small, increasing it should not prove problematic.

7.4 Language Issues

Several features of the Java language had a direction impact on the ease of implementation, debugging, optimising and testing. Like most language features, these represent particular trade-offs in terms of flexibility, speed, succinctness, ease of reading etc, however, there were a number of features which were sorely trying:

- The lack of access to parent packages by sub-packages. Sub-packages have no special access to their parent packages. If classes in a sub-package were able to access classes in the parent package as though they were of the parent package, sub-packages could have been used to separate the garbage collection from other memory management functions, without making the low-level memory management internals globally accessible. In particular the object MemoryObject is shared between both the garbage collection classes and memory manger classes. Java doesn't allow these two packages to share a package relationship with this object without either exposing themselves to each other as well (by merging the two packages) or making MemoryObject's internals public.

- The lack of ability to promote methods from non-static to static. In several classes, particular implementations provide methods which may be made static, but are not static in the general case. Java provide no way of promoting a method to static, which would allow optimisations not possible on non-static classes. For example AbstractCirclist defines a short next(short object) method. Snaplist implements AbstractCirclist, and it's implementation of next() is static (in that it doesn't use the

this pointer, implicitly or explicitly), but can't be declared as such. Java allows operator overloading only on the basis of parameter types, so declaring a second, static, version of the function with the same parameters is not permitted.

This problem will hopefully be solved by sufficiently advanced compilers, code generators and runtime systems which recognise that methods which don't access the this pointer are good candidates for certain types of optimisations.

- The lack of ability to determine the current class in a static method. When writing test methods, it is often useful to have a static function (for example main) create an instance of class and perform some operations on it to check it's correctness. To construct such an object, in the absence of a this object to query for the class, requires explicit specification of the class. When a derived class is created, and the test method inherited, the explicit specification still refers to the parent class. For example the Snaplist class (a doubly linked list) is extended by RSnaplist (which implements robustness), both of these classes have a main() which tests their functionality. main in Snaplist creates Snaplist a performs tests on it, ideally RSnaplist should be able to reuse this code with a call such as super.main() to ensure that the implementation of RSnaplist doesn't break any Snaplist features. main in Snaplist, however, has no way to determine that it should be working with a subclass.

  A work around to this problem is to declare test functions as accepting an instance of the class to be tested. The Java reflection classes within java.lang.reflect can then be used to dynamically create as many instances of this class as are necessary.

- The lack of ability to specify static methods and/or members in interfaces. Ideally, all classes featured in the class decomposition would have been Java interfaces—which may be implemented by any class in a system, rather than abstract classes—which must form a directed acyclic graph rooted with java.lang.Object, limiting implementation options. Unfortunately some design patterns (such as a the Singleton) are impossible to implement using interfaces because getInstance() *must* be static.

  A work around to this problem is to declare singletons as abstract classes rather than interfaces. This, however, leads to other problems as Java allows only single inheritance. A class may implement an arbitrary number of interfaces, but extend only a single class.

There were also, a number of features of the language which proved a boon implementation, debugging and testing:

- The type-safe casting system. The Java runtime throws an exception rather than perform an unsafe type cast. This both ensures type-safety and allows run-time typing of objects. Coming from C/C++ background, I found this method of error detection much easier to use than the combination of bus errors, segmentation faults and subtle bugs which result from incorrect casting in other languages on the same development platform (Solaris 2.5).

- The java.lang.Thread.dumpStack() method. This provides a safe, robust, transparent and non-terminating way of examining the execution stack. This is particularly useful when debugging classes and methods which have more than one caller to determine in which situations certain problems arise. Displaying the offending state and calling java.lang.Thread.dumpStack() appears to be the best general-purpose response on reaching an erroneous program state.

- The java.lang.Object.finalize() methods. In combination with frequent invocations of the (native) garbage collector, this allows prompt notification of non-reachability of objects. Testing using MemoryObject subclasses with finalize() overridden to display state, principally whether the application thinks the object is still reachable, it is possible to identify objects which are incorrectly identified as unreachable by the garbage collector, indicating bugs the the collector implementation. The idea of using garbage collectors to debug other memory managers is not new, the Boehm collector comes with step-by-step instructions for it's use in this manner.

7.5 Implementation Testing

7.5.1 Timing problems

Initially, it was intended to test the implementation by timing the length of time taken to perform various test routines and programs. Testing was performed using the Java internal java.lang.System.currentTimeMillis(). During testing, however, I experienced problems with the reproducibility of timing results. The problems fell into two categories:

- Changes to seemingly unrelated pieces of code resulted in performance differences in critical sections. While the source of this problem is not immediately clear, it is possible that optimisation decisions in both the Java compiler and Java virtual machine (JVM) were based on average-case assumptions which didn't hold on my platform or in my code.

Machine name	CPU type	CPU speed (MHz)	RAM (MBytes)	static (ms)	non-static (ms)
ra	SPARC-4	110	28	*61*	*49*
purau	SPARC-5	85	20	*89*	*77*
kiwi	SPARC-10/41	50	156	82	92
mohua	SPARC-10/41 x2	40	124	96	100
matata	SPARC-20/51	50	283	46	48
pukeko	Ultra-1/170	167	94	33	35

Table 7.2: Table showing the speed of two types of Java method calls on six different machines of varying type. The meanings of the machine names are as follows: 'ra' the Maori sun god; 'purau' a bay in Banks Peninsula; 'kiwi' a native bird and New Zealands' national symbol; 'mohua' the Maori word for yellowhead; 'matata' the Maori word for fernbird; 'pukeko' a native swamp bird.

- The relative speed of static and non-static calls varied widely from CPU to CPU, even within the same class of family of CPU's (SPARC) with the same JVM. For example Table 7.2 shows the relative speeds of 25000 trivial function calls for each of the two different function calls on four different machines. Each time represents the fastest of 100 repeats of the 25000 calls, all machines used the same version if the JVM (1.1.1), the same version of the compiled code (compiled code didn't vary according to the machine on which it was compiled).

  It should be noted, however, that these results could be repeated $\pm$ 1 milli-second either on the same machine, or on other machines with the same CPU type, but a different network connectivity, main memory size, screen and cpu-load.

  No logic has been found for static calls being slower than non-static on ra and purau, this seem especially strange given that a static call can be trivially converted to a non-static at compile-time.

Both of these run contrary to the mainstream thought and common practice in optimisation (Aho *et al.*, 1986; Dean *et al.*, 1995), and appear to be entirely implementation dependent—that is there is nothing stated or implied in the language or virtual machine specification which means they *must* perform in this manner, nor to suggest that they will in the next release of the Java reference implementation.

These results call into question the usefulness of measuring performance by timing, and raise questions about it's usefulness of a result, even if I could measure it. They certainly rule out timing as a tool when "tweaking" for performance.

To some extent these problems may be the result of both the Java compiler and the JVM being first generation, Java is a new language with quite different compiler

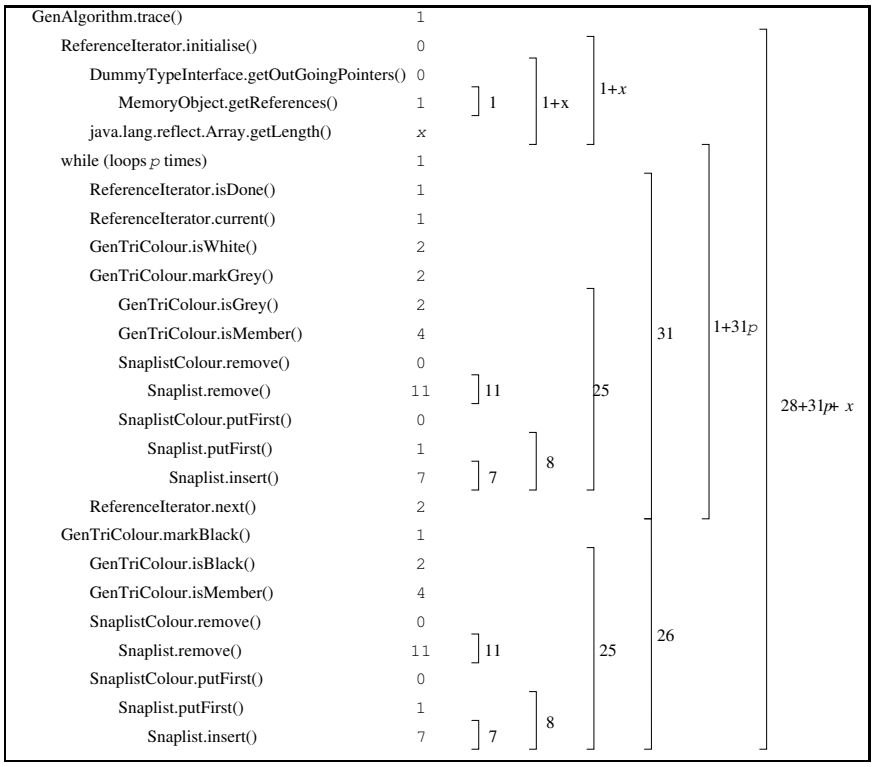


Table 7.3: Algorithmic analysis of GenAlgorithm.trace(), showing all method calls and looping structures. The method call java.lang.reflect.Array.getLength(), which is part of the Java runtime system, is a somewhat known factor. However, it's execution time (x) is likely to be very low as a call to this function (or an equivalent one) must be performed on every array reference, to check for array bounds errors. p is the maximum object complexity—the maximum number of references or pointers in an object.

and runtime requirements to both it's immediate predecessor, C++ , and other object-oriented languages such as Oberon, Smalltalk and Sather. As the language, it's compilers, optimisers and code transformers evolve hopefully the situation will become clearer.

7.6 Algorithmic Analysis

An alternative to runtime timing analysis is algorithmic analysis—the estimation of upper and lower bounds based on the algorithms used within an a system or subsystem. Timing of running programs can provide information such as how long on average and section of code takes to execute of the longest or shortest period of time taken for a particular trial, algorithmic analysis can make statements about absolute best- and worst-cases.

Algorithmic analysis performs best when the asymptotic or worst time complexities are considered (Kingston, 1990; Aho & Ullman, 1995), as is necessary in real-time systems,

GenCollector.writeBarrier()	0		
GenCollector.writeBarrier()	1		32+31p+x
GenAlgorithm.active()	1	31+31p+x	
GenTriColour.isGrey()	2		
GenAlgorithm.trace()	28+31p+x		

Table 7.4: Algorithmic analysis of GenCollector.writeBarrier(), showing all method calls and looping structures. See figure 7.3 for the detailed analysis of GenAlgorithm.trace(). p is the maximum object complexity. x is the time taken to call java.lang.reflect.Array.getLength().

because in systems of non-trivial complexity it is extremely hard to exercise all execution paths through the source, and even then ensuring the worst-case case been seen can be difficult (Ghosh *et al.*, 1993; Panzieri & Davoli, 1993).

The following algorithmic analysis is performed in the slightly more complex of the two implementations, GenCollector. The analysis of the code was greatly simplified by two facts

- no recursive calls are made—there are no recursive calls in any part of the collector
- the scarcity of looping constructs—in the analysis of GenCollector.writeBarrier() and GenCollector.register() presented here, there is one true loop (in GenAlgorithm.trace(), figure 7.3) and one section of repeated code (in GenAlgorithm.doQuanta(), figure 7.5), which is treated as a loop for the purposes of analysis.

Tables 7.3 and 7.4 show the cost of maintaining the write barrier, GenCollector.writeBarrier(), 32+31p+x. The x factor is the cost of a single method call to java.lang.reflect.Array.getLength(), which is implementation dependent, but almost certainly very low, since this method call (or one of equivalent functionality) must be called by the Java runtime system every time an array element is accessed to perform bounds checking. The p factor is complexity of the most complex heap object (that is the number of out going pointers or references). For the vast majority of objects, this complexity is known at compile time, however, allocations such as new Object[y] create arrays of objects of complexity y, assuming that arrays of objects are implemented as arrays of references to objects, or y multiplied by the complexity of object. It seems unlikely that the maximum object complexity for non-array objects in real-time systems would exceed 20–40, and the maximum size of arrays of pointer types 40-60. Because this is the only non-constant factor, GenCollector.writeBarrier() is $O(p)$.

Tables 7.5, 7.6, and 7.7 show the cost of registering a single object and performing the associated garbage collection. GenCollector.register() has a cost of not more than $440 + 84r + 186p + 3x$. x and p have the same definition as discussed previously. r is

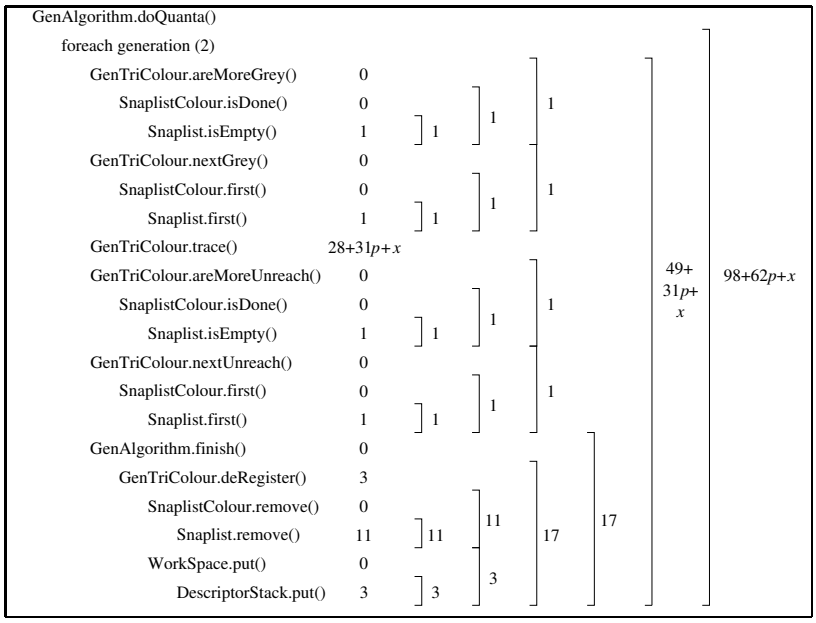


Table 7.5: Algorithmic analysis of GenAlgorithm.doQuanta() part A, showing all method calls and looping structures, showing the case where a flip is not performed. See table 7.3 for the detailed analysis of GenAlgorithm.trace(). p is the maximum object complexity. x is the time taken to call java.lang.reflect.Array.getLength().

the number of roots, the number of pointers into the heap from elsewhere. It is hard to imagine a precalculated upper bound for r. As before, the x factor is constant, making GenCollector.register() $O(r + p)$.

Similar analyses have been performed for the RootSet operations, however I believe that the LinkedListRootSet implementation of RootSet is unlikely to be used in a real system, as I believe an implementation with a more intimate knowledge of the workings of the execution stack could reduce addRoot() and removeRoot() costs to $O(0)$ while leaving the $O(r + p)$ of register() intact. Such efficiency savings appear possible by having an Iterator which iterates directly over the heap, "piggy-backing" on the normal stack frame pointers, rather than maintaining a separate data structure.

Table 7.8 shows the worst case performance of each of the collector algorithms and RootSet implementations. In each case, addRoot() and removeRoot() are inserts and removes in the respective data structures (see table 7.1).

The operations are given in table 7.8 in approximate order of frequency, and show that the RootSet remains a performance bottle-neck mainly removeRoot(). The $O(n)$ performance for external linked list rootsets is very pessimistic, in that the overwhelming majority of roots are from the stack, and stack-like operations on the linked list are

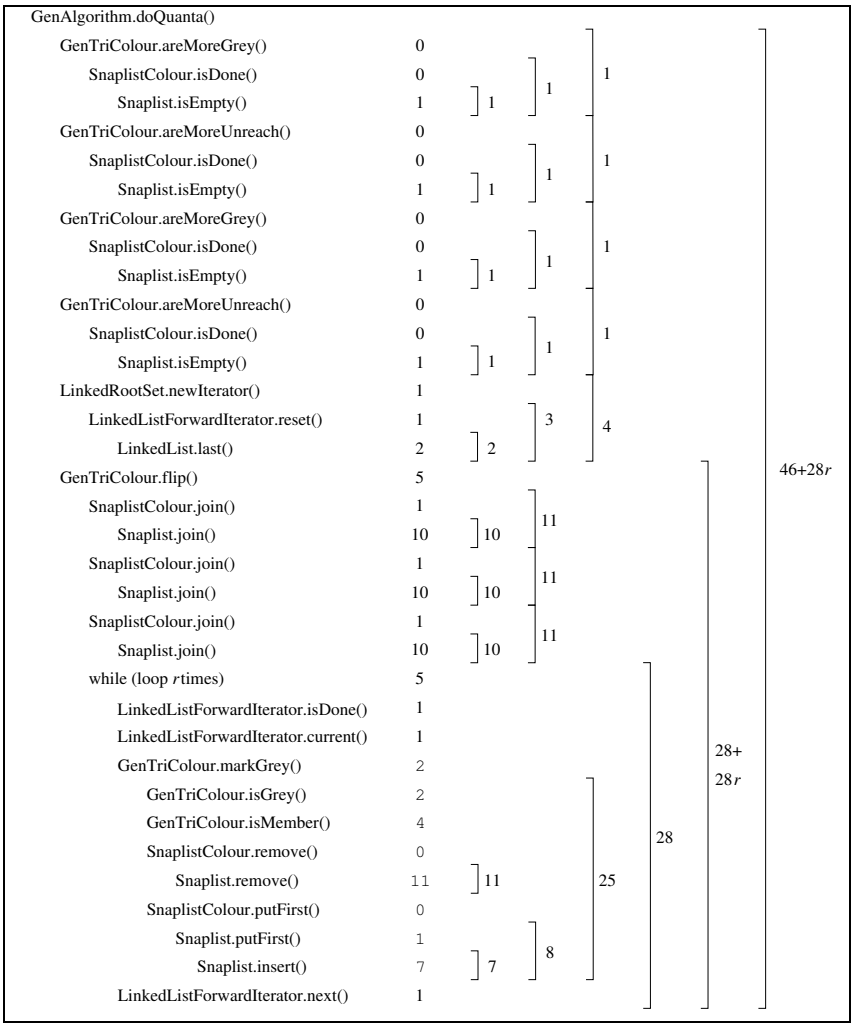


Table 7.6: Algorithmic analysis of GenAlgorithm.doQuanta() part B, showing all method calls and looping structures, showing the case where a flip is performed. r is the number of roots (pointers into the heap from the stack etc).

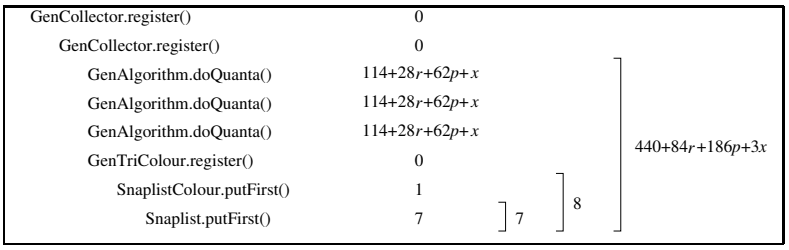


Table 7.7: Algorithmic analysis of GenCollector.register(), showing all method calls and looping structures. The cost of GenAlgorithm.doQuanta() was obtained by adding the costs of the two branches of the if-else (tables 7.5 and 7.6 respectively), this is an over-estimation. r is the number of roots (pointers into the heap from the stack etc). p is the maximum object complexity. x is the time taken to call java.lang.reflect.Array.getLength().

Algorithm	addRoot()	removeRoot()	writeBarrier()	register()
Generational	$O(1)$	$O(r)$	$O(p)$	$O(p+r)$
Mark-and-Sweep	$O(1)$	$O(r)$	$O(p)$	$O(p+r)$

Table 7.8: The worst case performance characteristics for both implemented algorithms. Where r is the maximum number of roots, p is the maximum number of pointers in a heap object.

performed in $O(1)$ time.

7.7 Summary

The garbage collector designed using the patterns described in chapter 6 for a realtime use. Appropriate algorithms were selected from those described in chapter 2. The collector was the implemented using what appeared to be the most appropriate data structures.

The performance of the resulting garbage collector was then assessed using algorithmic analysis. Described in terms of r, the maximum number of roots and p, the maximum number of pointers in a heap object, performance was found to be satisfactory for all operations but maintenance of the RootSet. A discussion of this and possible solution was presented.

Chapter 8

Conclusion

> Today, there is a vast gulf between the dynamic random-access memory chip (DRAM) provided by the hardware designer and the dynamic object-oriented graph structure desired by the software designer. This gulf must be filled by memory management hardware and software (Baker, 1995a)

8.1 Implementation

The two main surprises while implementing the garbage collectors were the importance of iterators and the number of levels of indirection.

The two Iterators I implemented, LinkedList and Snaplist, are substantially bigger than any of the classes used in the Mark-and-Sweep collector, both in terms of source code and object code, their method calls are featured in inner loops of all the algorithmic analyses I performed. They are an obvious target for performance tuning.

No more than a cursory browse of the source code, or algorithmic analyses, should be necessary to realise that many of the method calls are nothing but a level of indirection. This effect is not limited to my collector, many functions of the Boehm collector are implemented as #define macros, many of which are in excess of twenty lines, to overcome these problems. I used Java's 'inner classes' to achieve a similar effect. LinkedList, for example, contains the LinkedListBackwardIterator and LinkedListForwardIterator classes. Because inner classes have limited visibility and are in the same compilation unit as the data structures they operate on, compilers have increased opportunity to perform local optimisations.

The Mark-and-Sweep collector (MSCollector, MSAlgorithm and MSTriColour) appears to work, and appears robust. While I have not performed testing on 'real' applications, I have tested many permutations of collector state, I have confidence in the collectors

correctness.

The Generational collector (GenCollector, GenAlgorithm and GenTriColour) appears to have an bug. The bug is almost certainly in GenTriColour, possibly in GenTriColour.flip(). In hindsight, making the simplification of a two generation generational collector proved not such a simplifying assumption, amounting to more than the loop unrolling I anticipated. The current implementation uses a class which permits only two instances to be created, which then share certain aspects of state.

A better design may have been to split the responsibility for managing the bit patterns for use in MemoryObject._tag off into a separate singleton class and eliminating sharing of state between GenTriColour instances.

Both collectors share a RootSet implementation which is to slow. It appears that this may be rectified by implementing RootSet as an iterator over the runtime stack, eliminating the need to maintain a secondary data-structure mirroring it's functionality. This, would, however, substantially increase coupling between the garbage collector and other components of the runtime system.

8.2 Final Remarks

My main original contribution is the design patterns documented in chapter 6, in particular TriColour, which represents methodology with a 30-year documented history in computer science which as, hitherto, not been captured as a design pattern.

8.3 Further Work

There are several directions in which the work outlined in this thesis could be continued:

- An examination of a homogeneous group of garbage collectors for design patterns would be an interesting comparison to the relatively heterogeneous group I examined. This would shed light on the degree to which different implementation choices are linked to language and usage issues, and clarify systems in which the various design patterns are applicable.
- The implementation of a more efficient RootSet would be desirable. This, however, isn't so much a garbage collection problem but the runtime stack providing a Iterator and the garbage collector providing a RootSet Adaptor for it.
- A redesign of the generational collector with the separate singleton class for managing the bit patterns.

Acknowledgements

I'd like to thank: Michel de Champlain for his assistance, guidance and perseverance; my father for his proof-reading of, and comments on, this thesis; my brother for his support during the early part of the thesis; my fellow postgrads, ex-postgrads and the support staff, for helpful comments, discussion and camaraderie; the coven and friends for their unflagging support, encouragement and offers of food; my lablings, whose demand for clarity of answers clarified my own thoughts; my fellow tutors who demonstrated that clarity of thought, and explanation, was achievable; the computer science department at the University of Waikato for use of their library and facilities during the summer of '95–'96 and finally the computer science department at the University of Canterbury for the veritable well-spring of caffeinated drinks.

Bibliography

Aarsten, Amund, Brugali, Davide, & Menga, Giuseppe. 1996. Designing concurrent and Distributed Control Systems. *Communications of the ACM*, **39**(10), 50–57. Design patterns issue.

Aho, Alfred V., & Ullman, Jeffrey D. 1995. *Foundations of Computer Science*. C edition edn. Freeman.

Aho, Alfred V., Sethi, Ravi, & Ullman, Jeffrey D. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Appel, Andrew W., & Shao, Zhong. 1994 (March 8). *An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures*. Tech. rept. CS-TR-450-94. Princeton University.

Atkins, Martin C., & Nackman, Lee R. 1988. The Active Deallocation of Objects in Object-Oriented Systems. *Software—Practice and Experience*, **18**(11), 1073–1089.

Baker, Henry G. 1978. List processing in Real Time on a Serial Computer. *Communications of the ACM*, **21**(4), 280–294.

Baker, Henry G. 1992. The Treadmill: Real-time Garbage Collection Without Motion Sickness. *SIGPLAN Notices*, **27**(3), 66–69.

Baker, Henry G. 1995a. Editorial. *In:* (Baker, 1995b).

Baker, Henry G. (ed). 1995b. *IWMM95*. Lecture Notes in Computer Science, no. 986. Springer.

Bekkers, Y., & Cohen, J. (eds). 1992. *IWMM92*. Lecture Notes in Computer Science, no. 637. Springer.

Boehm, Hans-Juergen. 1993. Space Efficient Conservative Garbage Collection. *SIGPLAN Notices*, **28**(6), 197–206.

Boehm, Hans-Juergen, & Weiser, Mark. 1988. Garbage Collection in an Uncooperative Environment. *Software—Practice and Experience*, **18**(9), 807–820.

Boehm, Hans-Juergen, Demers, Alan J., & Shenker, Scott. 1991 (June). Mostly Parallel Garbage Collection. *Pages 157–164 of: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation.* ACM.

Booch, Grady. 1994. *Object-oriented analysis and design with applications.* Addison-Wesley.

Buschmann, Frank, & Meunier, Regine. 1995. Pattern Languages of Program Design. *In:* (Coplien & Schmidt, 1995).

Buschmann, Frank, Meunier, Regine, Robnert, Hans, Sommerlad, Peter, & Stal, Michel. 1996. *Pattern-Oriented Software Architecture: A system of patterns.* Wiley.

Cline, Marshall P. 1996. The Pros and Cons of Adopting and Applying Design Patterns in the Real World. *Communications of the ACM*, **39**(10), 47–49. Design patterns issue.

Clinger, William, & Rees, Jonathan. 1987. *Revised^4 Report on the Algorithmic Language Scheme.*

Collins, George E. 1960. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, **3**(12), 655–657.

Cooper, Doug. 1990. *Oh My! Modula-2!* Norton.

Coplien, James O., & Schmidt, Douglas C. (eds). 1995. *Pattern Languages of Program Design.* Addison-Wesley.

Corporaal, H. 1991a. Automatic Heapmanagement and Realtime Performance. *Pages 290–295 of: Proceedings, Advanced Computer Technology, Reliable Systems and Applications, 5th annual European Computer conference 91.* IEEE.

Corporaal, H. 1991b. Real realtime performance of heapmanagement systems, using incremental copying collectors. *Pages 344–352 of: Proceedings of the Twenty Fourth Annual Hawaii International Conference on System Sciences*, vol. 1. IEEE.

Corporaal, H., & Veldman, T. 1991. The Design Space of Garbage Collection. *Pages 423–428 of: Proceedings, Advanced Computer Technology, Reliable Systems and Applications, 5th annual European computer conference.*

Corporaal, H., Veldman, T., & van de Goor, A. J. 1990. An efficient, Reference Weight-based Garbage Collection Method for Distributed Systems. *Pages 463–465 of: PARBASE-90 International Conference on Databases, parallel Architectures and their Applications.* IEEE.

de Champlain, M. 1995. *Modèle réactif and réflexif pour méta machines à états finis.* Thèse Ph.D., Département de génie électrique and de génie informatique, Ecole Polytechnique de Montréal, Université de Montréal, Montréal, QC, Canada.

de Champlain, M. 1996a. A Pattern Language for Porting Micro-kernels. *Proc. of the 5th Annual International Workshop on Object Orientation in Operating Systems (IWOOOS '96), Seattle, WA*, October, 144–150.

de Champlain, M. 1996b. Patterns to Ease the Port of Micro-kernels in Embedded Systems. *Proc. of the 3rd Annual Conference on Pattern Languages of Programs (PLoP'96), Allerton Park, IL*, June.

Dean, Jeffrey, Grove, David, & Chambers, Craig. 1995 (August). Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. *Pages 75–101 of: ECOOP95.*

Deutsch, L. Peter, & Bobrow, Daniel G. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Transactions of the ACM*, **19**(9), 522–526.

Dijkstra, Edsger W., Lamport, Leslie, Martin, A. J., Scholten, C. S., & Steffens, E. F. M. 1978. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, **21**(11), 966–975.

Epstein, Richard L., & Carnielli, Walter A. 1989. *Computability: computable functions, logic and the foundations of mathematics.* Wadsworth and Brooks.

Fenichel, Robert R., & Yochelson, Jerome C. 1969. A LISP Garbage Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, **12**(11), 611–612.

Ferreira, Paulo. 1991. Reclaiming storage in an object oriented platform supporting extended C++ and Objective-C applications. *Pages 100–102 of: Proceedings of the 1991 International workshop on Object Orientation in Operating Systems.* IEEE.

Finkel, Raphael A. 1995. *Advanced Programming Language Design.* Addison-Wesley.

Gabriel, Richard P. 1996. *Patterns of software: tales from the software community.* Oxford University Press.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. *In: ECOOP'93 Object-Oriented Programming.*

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Ganesan, Ravichandran. 1994. *Local Variable Allocation For Accurate Garbage Collection of C++.* Tech. rept. Iowa State University.

Gao, Hong, & Nilsen, Kelvin D. 1994 (July). *Reliable General Purpose Dynamic Memory management for Real-Time Systems.* Tech. rept. TR94-09. Iowa State University.

George O. Collins, Jr. 1961. Experience in Automatic Storage Allocation. *Communications of the ACM*, **4**(11), 436–440.

Ghosh, Kaushik, Mukherjee, Bodhisattwa, & Schwan, Karsten. 1993 (February). *A survey of Real-Time Operating Systems—Draft.* Tech. rept. GIT-CC-93/18. College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 30332-0280.

Goldberg, Benjamin. 1991. Tag-Free Garbage Collection for Strongly Typed Programming Languages. *Pages 165–176 of: Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

Gosling, James, Joy, Bill, & Steele, Guy. 1996. *The Java Language Specification.* Addison-Wesley.

Guggilla, Satish Kumar. 1994. *Generational Garbage Collection of C++ Targeted to SPARC Architectures.* Tech. rept. Department of Computer Science, Iowa State University.

Hamilton, G. W. 1995. Compile-Time Garbage Collection for Lazy Functional Languages. *In:* (Baker, 1995b).

Harbaugh, Sam, & Wavering, Bill. 1991. HeapGuard^{TM}, Eliminating Garbage Collection in Real-Time Ada Systems. *Pages 704–708 of: Proceedings of the IEEE 1991 National Aerospace and Electronics Conference NAECON 1991*, vol. 2.

Heeb, Beat, & Pfister, Cuno. 1990?. *Oberon Technical Notes.*

ISO9899. 1990. ISO-9899—Harmonized standard for the C programming language.

JDK1.0.2. 1996 (May). *Documentation accompanying the Java Development Kit version 1.0.2*. Sun Microsystems.

JDK1.1.1. 1997 (Feb). *Documentation accompanying the Java Development Kit version 1.1.1*. Sun Microsystems.

Jones, Richard, & Lins, Rafael. 1996. *Garbage Collection:Algorithms for Automatic Dynamic Memory Management*. Wiley.

Jones, Simon B., & Tyas, Andrew S. 1994 (September). *The Implementor's Dilemma: A Mathematical Model of Compile Time Garbage Collection*. Tech. rept. CSM-118. University of Stirling, Stirling, Scotland.

Jonkers, H. B. M. 1983. *Abstraction, Specification and Implementation Techniques with an application to garbage collection*. Mathematical Centre Tracts, no. 166. Amsterdam: Mathematisch Centrum.

Kernighan, Brian W., & Ritchie, Dennis M. 1988. *The C Programming Language*. Second edition edn. Prentice Hall.

Kingston, Jeffrey H. 1990. *Algorithms and Data Structures—Design, Correctness, Analysis*. Addison-Wesley.

Kofler, Thomas. 1993. Robust Iterators in ET++. *Structured Programming*, **14**(March), 62–85.

Kordale, R., Ahamad, M., & Shilling, J. 1993. Distributed/Concurrent Garbage Collection in Distributed Shared Memory Systems. *Pages 51–60 of: Proceedings of the Third International Workshop on Object Orientation in Operating Systems*.

Krämer, Christian, & Prechelt, Lutz. 1996 (8–10 November). Design Recovery by Automated Search for Structure Design Patterns in Object-Oriented Software. *Pages 208–215 of: Proceedings of Working Conference on Reverse Engineering 1996*. IEEE, Monterey, CA.

Kuechlin, Wolfgang W., & Nevin, Nicholas J. 1991. On Multi-threaded List-Processing and Garbage Collection. *Pages 894–897 of: Proceedings of the Third IEEE Symposium on Parallel and distributed Processing*.

Kuo, Wen-Yan, & Kuo, Sy-Yen. 1993. Parallel Garbage Collection and Graph Reducer. *Pages 100–103 of: Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*.

LaLonde, Wilf, & Pugh, John. 1994. *Smalltalk V Practice and Experience.* Prentice Hall.

Lea, Doug. 1997. *Concurrent Programming in Java: Design Principles and Patterns.* SunSoft/Addison-Wesley.

Lindholm, Tim, & Yellin, Frank. 1996. *The Java Virtual Machine Specification.* Addison-Wesley.

Lins, Rafael D. 1992. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, **44**(4), 215–220.

Magnusson, Boris, & Henriksson, Roger. 1995a. Garbage Collection for Control Systems. *In:* (Baker, 1995b).

Magnusson, Boris, & Henriksson, Roger. 1995b (August). Garbage Collection for Hard Real-Time Systems. *Pages 60–63 of: Proceeding. Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS '95).*

McCarthy, John. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, **3**(4), 184–195.

McKenney, Paul E. 1996. Selecting Locking Primitives for Parallel Programming. *Communications of the ACM*, **39**(10), 75–82. Design patterns issue.

Meyer, Bertrand. 1994. *Reusable Software: The Base object-oriented component libraries.* Prentice Hall.

Mohnen, Markus. 1995 (May 9). *Efficient Compile-Time Garbage Collection for Arbitrary Data Structures.* Postscript document.

Morris, F. Lockwood. 1978. A Time- and Space-Efficient Garbage Compaction Algorithm. *Communications of the ACM*, **21**(8), 662–665.

Mössenböck, Hanspeter. 1993. *Object Oriented Programming in Oberon-2.* Springer. Translated (from German to English) by Robert Bach. Forword by Niklaus Wirth.

Murray, Robert B. 1993. *C++ Strategies and Tactics.* Professional Computing Series. Addison-Wesley.

Nilsen, Kelvin D. 1994. Reliable Real-Time Garbage Collection of C++. *Computing Systems*, **7**(4), 467–503.

Nilsen, Kelvin D. 1995 (November 15). *Issues in the Design and Implementation of Real-Time Java.* Tech. rept. Ohio State University.

Nilsen, Kelvin D., & Gao, Hong. 1995. *The Real-Time Behavior of Dynamic Memory Management in C++.* Tech. rept. Ohio State University.

Ousterhout, John K. 1994. *Tcl and the Tk toolkit.* Addison-Wesley.

Pagel, Bernd-Uwe, & Winter, Mario. 1996. Towards Pattern-Based Tools. *Pages n–n+11 of: EuroPlop96.*

Panzieri, F., & Davoli, R. 1993 (Oct.). *Real Time Systems: A Tutorial.* Tech. rept. Laboratory for Computer Science, University of Bologna.

Plainfossé, David, & Shapiro, Marc. 1995. A Survey of Distributed Garbage Collection Techniques. *In:* (Baker, 1995b).

Rada, Roy. 1995. *Software Reuse: Principles, Methodologies and Practices.* Oxford, England: Intellect.

Sanaran, Nandakumar. 1994 (February). *A Bibliography on Garbage Collection.* Tech. rept. Computer Science Department, Clemson University. ftp://ftp.cs.clemson.edu/techreports/94-102.ps.Z.

Sane, Aamod. 1995 (December 14). *The Elements of Pattern Style.* Draft: derived from discussion sessions of PLOP95.

Seligmann, Jacob, & Grarup, Steffen. 1995. Incremental Mature Garbage Collection Using the Train Algorithm. *In: Proceedings of ECOOP '95—9th European Conference on Object Oriented Programming.* Springer.

Steele, Jr., Guy L. 1975. Multiprocessing Compactifying Garbage Collection. *Communications of the ACM*, **18**(9), 495–509. Awarded first place in the ACM 1975 George E. Forsythe Student Paper Competition Awards.

Stroustrup, Bjarne. 1986. *The C++ Programming Language.* second edn. Addison-Wesley series in computer science. Reading, Massachusetts: Addison-Wesley.

Sun. 1995. *The Java Language Specification.* 1.0 alpha 3 edn. Sun Microsystems.

SunMan. 1995. *Solaris 2.5 Reference Manual AnswerBook.* Sun Microsystems, Mountain View, CA, USA.

Tanaka, Yoshio, Matsui, Shogo, Maeda, Atsushi, & Nakanishi, Maskazu. 1994. Partial Marking GC. *Pages 337–348 of: Parallel processing : CONPAR 94-VAPP VI : third Joint International Conference on Vector and Parallel Processing.* Springer.

Tanenbaum, Andrew S. 1992. *Modern Operating Systems.* Prentice-Hall.

Uhl, Jürgen, & Schmid, Hans Albrecht. 1990. *A Systematic Catalogue of Reusable Abstract Data Types.* Lecture notes in Computer Science, no. 460. Springer.

Ungar, David. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *SIGPLAN Notices*, **19**(5), 157–167.

van Mierlo, Marcel. 1996 (March 11). *Layout for Survey of Garbage Collection Design and Implementation issues.* Report of research conducted over the 1995-96 summer.

Virding, Robert. 1995. A Garbage Collector for the Concurrent Real-Time Language Erlang. *In:* (Baker, 1995b).

Washabaugh, Douglas M., & Kafura, Dennis. 1990. Incremental Garbage Collection of Concurrent Objects for Real-Time Applications. *Pages 21–30 of: Proceedings of the 11th Real-time Systems Symposium.*

Weide, B. W., Edwards, S. H., Harms, D. E., & Lamb, D. A. 1994. Design and Specification of Iterators Using the Swapping Paradigm. *IEEE Transactions on Software Engineering*, **20**(8), 631–643.

Wentworth, E. P. 1990. Pitfalls of Conservative Garbage collection. *Software—Practice and Experience*, **20**(7), 719–727.

Wilson, Paul R. 1992. Uniprocessor Garbage Collection Techniques. *In:* (Bekkers & Cohen, 1992).

Wilson, Paul R., Lam, Michael S., & Moher, Thomas G. 1991 (June). Effective "static-graph" Reorganisation to Improve Locality in Garbage-Collected Systems. *Pages 177–191 of: Proceedings of ACM SIGPLAN 1991.*

Wilson, Paul R., Johnstone, Mark S., Neely, Michael, & Boles, David. 1995. Dynamic Storage Allocation: A Survey and Critical Review. *In:* (Baker, 1995b).

Wirth, Niklaus, & Gutknecht, Jürg. 1990. *Project Oberon: The Design of an Operating System and Complier.* New York, New York: ACM Press.

Yip, May. 1991 (May). *Incremental, generational mostly-copying garbage collection in uncooperative environments.* M.Phil. thesis, Electrical Engineering and Computer Science, MIT. ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TR-91.8.ps.

Zimmer, Walter. 1995. Pattern Languages of Program Design. *In:* (Coplien & Schmidt, 1995).

Chapter A

Glossary

Accurate — an algorithm which does not over-estimate live objects. Implies complete type information. Compare **Conservative**.

Collector — a thread of control or a system performing garbage collection. Compare **Mutator**.

Conservative — an algorithm which over-estimates live objects. Generally caused by lack of type information or action of the read- or write-barrier. Compare **Accurate**.

Copying garbage collection — garbage collection algorithms in which reachable objects are copied each cycle.

Destructor — a method called immediately before memory for an object is reclaimed. A synonym of **Finaliser**.

External data structure — a data structure in which the space overhead for the structure is separate from the atoms of the structure, for example a linked list using a **Node** class in point to the present atom and next **Node**, and which could handle atoms of any class. Compare **Internal data structure**.

Finaliser — a method called immediately before memory for an object is reclaimed. A synonym of **Destructor**.

Fragmentation — scattering of heap objects through memory resulting in unused, but unavailable memory. Increases an programs memory requirements.

Generational garbage collection — similar to Semi-Space garbage collection algorithms, with an additional space to hold old, long lived objects .

Incremental garbage collection — garbage collection algorithms in which collector activity is mingled with mutator activity.

Internal — a data structure in which the space overhead for the structure is within the atoms of the structure, for example a linked list whose elements are all subclasses of a **ListItem** class which contained a pointer to the next item. Compare **External data structure**.

Live — an object is live if it will be accessed by the mutator in the future. Compare **Reachable** and **Unreachable**.

Mark-and-Sweep garbage collection — garbage collection algorithms in which all items reachable form the roots are marked, and then in a separate pass over the heap, all unmarked items are removed.

Mutator — a thread of control, or subsystem, performing the central task of the application. Compare **Collector**.

Non-copying garbage collection — garbage collection algorithms in which unreachable objects are placed on a free list for future use.

Non-incremental garbage collection — garbage collection algorithms in which no mutator activity occurs during a garbage collection cycle.

Parallel garbage collection — garbage collection algorithms in which at least some of the collector activity proceeds in parallel with mutator activity.

Pointerful — an object which contains references or pointers to other objects. A node in a binary tree is typically pointerful object, an object storing a bitmap image is typically not.

Reachable — an object is reachable if it can be accessed by the mutator by any valid sequence of future actions. Compare **Live** and **Unreachable**.

Read-Barrier — a check, invoked on every read from the heap, used to maintain the tri-colour invariant. Often used in incremental copying collectors. Compare **Read-Barrier**.

Reference Counting — garbage collection algorithms in a count of inward references is kept for each item. When when the count reaches zero, the item is garbage. Compare **Sweeping**.

Semi-Space garbage collection — garbage collection algorithms in memory is divided into two parts, and reachable objects copied from one to the other each cycle.

Sweeping garbage collection — garbage collection algorithms which traverse or sweep the heap to determine chunk reachability. Compare **Reference Counting**.

Type Safe — a property of a language which prevents the user from casting between types. For example the C/C++ assignment `int a = (int) aPointer` is unsafe, as are untyped unions.

Unreachable — an object is unreachable if it can not be accessed by the mutator by any valid sequence of future actions. Compare **Reachable** and **Live**.

Write-Barrier — a check, invoked on every write to the heap, used to maintain the tri-colour invariant. Often used in incremental non-copying collectors. Compare **Read-Barrier**.

Chapter B

Resistance to Interference

B.1 Introduction

Modula-2 and C both have 'for' loops, but they differ in a very important way. The loop counter in Modula-2 may not be changed within the loop, whereas that in C may be changed. The latter represents probably the most simple case of interference—a control structure is set up, which then may be altered from as a side effect of other actions. Language rules about this vary, from prohibition (Modula-2 (Cooper, 1990)) or encouraging (C (Kernighan & Ritchie, 1988), C++ (Stroustrup, 1986), Java (Gosling *et al.*, 1996)) to the extreme case of interpreted languages such as tcl (Ousterhout, 1994), which permit not only the value of the loop counter but also it's scope to be manipulated with the loop.

All these languages are *robust*[1], their permissiveness in such situations, and the results of such manipulations, are clear from their specifications, allowing quality compilers and interpreters to issue warnings or errors as necessary.

With the generalisation of a 'for' loop to an arbitrary iteration over a set of values, either calculated numbers or atoms in a data structure or container object, the situation becomes considerably less clear. It is very easy in most languages to specify and design iterators over containers which have undefined semantics if other operations are performed on the container during the iteration. It is also relatively easy, although uncommon, to specify and design containers in such languages in a manner such that their iterators are robust (Weide *et al.*, 1994; Murray, 1993). Iterators in the NIHCL and Borland libraries for C++ "are not robust in any way" (Kofler, 1993). The Objective-C, Dee, CLU and Smalltalk-80 libraries are not robust. Eiffel offers limited robustness using 'cursors' which

[1] Robustness has other components, such as freedom from arbitrary limits, fault tolerance, etc. but resistance to interference is the only component I shall consider here.

permit only one of which is active on a list at once. However, SETL allows robust iteration using the languages' strict value semantics. (Kofler, 1993) contains an in-depth discussion of these.

An example of application logic which might cause interference problems is an attempt to code "ask each postgrad enrolled in the department to notify the secretary of other students who don't turn up so they can be dis-enrolled." If the first student queried notifies the secretary of a second student, and that student is promptly dis-enrolled, the iterator over each postgrad in the department may enter undefined state when it comes to iterate over the second student. Unless the iterators being used are robust, or aware of each other, it is not clear whether the second student will be iterated over or not, or indeed whether the program will crash.

The core problems are (1) global knowledge is required about the usage of non-robust objects to prove their correctness and (2) common non-robust libraries are not specified in a manner which allows automatic detection and notification of dubious, undefined or erroneous usage, putting the load of proving their correct usage into the programmers rather than their tools.

B.2 Implementing iterators resistant to interference

Robust iterators are may be implemented by shallow coping of containers and iteration over only private copies, or by registering iterators with the container and adjusting their state when the containers state changes. Both of these have extra overhead when the iteration is started and completed (copying or registration and freeing or deregistration), but only the registration technique also has overhead when the container is modified. Of the iterators I have written for my garbage collector:

- LinkedListForwardIterator LinkedListBackwardIterator and ReferenceIterator are non-robust, but this is not a problem because local knowledge was used to determine that the containers they iterated over were not modified during the iteration.

- SnaplistColour uses a stack-like method of semi-robust iteration, as local knowledge was used to determine that objects are always removed after being iterated over.

- RSnaplist extends Snaplist to allow robust iteration using the registration method. RSnaplist is not used in the garbage collector, because it wasn't strictly necessary and more efficient options were available. Several different iterators are available over RSnaplist:

- StubRSnaplistIterator, a base class, which provides no robustness features, but implements methods to interact with RSnaplist.
- NormalRSnaplistIterator a robust iterator which merely guarantees not to reach an invalid state if arbitrary numbers of objects are inserted or removed during the iteration.
- CompleteRSnaplistIterator a robust iterator which guarantees not to reach an invalid state and which guarantees that if isDone() returns true all objects in the object have been iterated over, but may iterate over the same object twice.

RSnaplist is implemented using fixed size arrays rather than a more flexible structure to avoid the possibility of object allocation, as mandated by the requirements (see section 7.1).

B.3 Robustness and Object Orientation

Encapsulation is a key part of object-orientation (Booch, 1994). In this section I argue that lack of robustness in publicly visible objects is a breach of encapsulation, hence robustness in globally visible objects is a requirement for object orientation.

Information as to whether an object has is currently being iterated over or not (and hence whether or not it is open to the full range of manipulations) may either be managed within the object, elsewhere, or or not at all. If the information is managed elsewhere, the information must have at least the same visibility as the object. If the object is in the global scope, then this information must be global, which is a breach of encapsulation. If the information is not managed by the system at all (as is typically happens in non-robust libraries) then the programmer must manage this information.

Use of local data to enforce correct library usage is be sufficient for programmer directed detection of incorrect in small systems. However, in the general case, use of such data in global situations is a breach of encapsulation and not scalable.

Working out whether a container object has an active iterator over it is similar to working out whether an object has is live in the garbage collection sense (see section 2.2):

- Both are easy to work out in 'simple' systems, but when references to objects are passed between subsystems exposing object life-cycle information represents a breach of encapsulation.
- Determining whether an iterator is active, or whether a reference to an object will be dereferenced, is a halting equivalent problem, so conservative assumptions must

be made.

- Simple reference counting schemes are flawed in the general case. Reference counting garbage collectors fail to reclaim cycles. A naïve implementation using read-/write-locks to post-fit robustness to container classes would appear to cause a deadlock on the postgrad example given earlier.

I see no resolution to these problems but the use of objects which are resistant to interference in for publicly accessible objects.

Chapter C

Source Code

C.1 Garbage Collection Implementation

C.1.1 Collector (facade)

```
package openKernel.objectManager;
/**
 * Interface Collector, the interface between the Garbage Collector
 * and the rest of the system.
 *
 * Unfortunately, becuase a derived class is a singleton, Collector
 * must be an abstract class, not an interface.
 *
 * @version  0.2, june 1997
 * @author   stuart yeates                                          10
 */

abstract class Collector {

  /** The singleton instance of this class */
  static Collector collector = null;

  /**
   * Get the singleton instance of this class
   * @returns the instance                                          20
   */
  static public Collector getInstance(){
    if (collector != null)
      return collector;
    else
      throw new java.lang.Error("Collector.getInstance() must be called " +
                                "for the first time via a derived class");
  };

  /**                                                               30
   * Register a freshly created heap object.
   * @param object the object to be added
   */
  abstract void register(MemoryObject object);
```

```
  /**
   * Notify the collector of a new root (pointer from outside the
   * heap into the heap).
   * @param root the root to be added
   */
  abstract void addRoot(MemoryObject root);

  /**
   * Notify the collector of the removal of a root
   * @param root the root to be removed
   */
  abstract void removeRoot(MemoryObject root);

  /**
   * Get an iterator over the rootSet
   */
  abstract Iterator getRoots();
  /**
   * Perform a small amount of garbage collection
   */
  abstract void doQuanta();

  /**
   * Perform a large amount of garbage collection
   */
  abstract void completeCollection();

  /**
   * Maintain the Read-Barrier. Called when the Application is
   * about to read object i. Not required if writeBarrier() is
   * being used.
   */
  abstract void readBarrier(short i);

  /**
   * Maintain the Write-Barrier. Called when the Application is
   * about to write to a field in object i. Not required if
   * readBarrier() is being used.
   */
  abstract void writeBarrier(short i);

  /**
   * Is it necessary to use the barriers at the moment ?
   */
  abstract boolean active();
}
```

C.1.2 Algorithm

```
package openKernel.objectManager;
/**
 * abstract class Algorithm, the driving algorithm of the garbage collector
 *
```

```
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

abstract class Algorithm {
                                                                    10
  /**
   * Register a freshly created heap object.
   * @param object the object to be added
   */
  abstract void register(MemoryObject object);

   /**
   * Perform a small amount of garbage collection
   */
  abstract void doQuanta();                                         20

  /**
   * Perform a large amount of garbage collection
   */
  abstract void completeCollection();

  /**
   * Register a freshly created root
   * @param object  the new root
   */                                                               30
  abstract public void newRoot(MemoryObject object);
  /**
   * Maintain the Read-Barrier. Called when the Application is
   * about to read object i. Not required if writeBarrier() is
   * being used.
   */
  abstract void readBarrier(short i);

  /**
   * Maintain the Write-Barrier. Called when the Application is        40
   * about to write to a field in object i. Not required if
   * readBarrier() is being used.
   */
  abstract void writeBarrier(short i);

  /**
   * Is it necessary to use the barriers at the moment ?
   */
  abstract boolean active();
}                                                                   50
```

C.1.3 TriColour

```
package openKernel.objectManager;
/**
 * Interface TriColour, the interface between the garbage collection
 * algorithm and the garbage collection state.
 *
```

```
 * White objects are of unknown reachability.
 *
 * Grey objects are reachable, but have pointers which have not been
 *   examined. they represent the current fringe of the traversal of
 *   the collector through the heap graph.                              10
 *
 * Black objects are reachable, and finished with.
 *
 * UnreachableObjects are those which are known to be unreachable
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

public interface TriColour {                                            20

  /**
   * is this object in this TriColour ?
   * @return true if object is grey
   * @param object  the object under consideration
   */
  boolean isMember(MemoryObject object);

  /**
   * is this object marked grey ?                                       30
   * @return true if object is grey
   * @param object  the object under consideration
   */
  boolean isGrey(MemoryObject object);

  /**
   * is this object marked white ?
   * @return true if object is white
   * @param object  the object under consideration
   */                                                                   40
  boolean isWhite(MemoryObject object);

  /**
   * is this object marked black ?
   * @return true if object is black
   * @param object  the object under consideration
   */
  boolean isBlack(MemoryObject object);

  /**                                                                   50
   * mark an object grey
   * @param object the object to be marked
   */
  void markGrey(MemoryObject object);

  /**
   * mark an object white
   * @param object  the object to be marked
   */
  void markWhite(MemoryObject object);                                  60
```

```
  /**
   * mark an object black
   * @param object  the object to be marked
   */
  void markBlack(MemoryObject object);

  /**
   * are there more grey objects ?
   * @return true if there are more grey objects                    70
   */
  boolean areMoreGrey();

  /**
   * which is the next grey item ?
   * @return the next grey MemoryObject
   */
  MemoryObject nextGrey();

  /**                                                               80
   * are there more unreachable objects ?
   * @return true if there are more unreachable objects
   */
  boolean areMoreUnreach();

  /**
   * which is the next unreachable item ?
   * @return the next unreachable MemoryObject
   */
  MemoryObject nextUnreach();                                       90

//  /**
//   * get a list of unreachable objects. Not even passingly useful if
//   */
//  Iterator getUnreachObjects();

  /**
   * start a new garbage collection cycle
   */
  void flip(Iterator rootset);                                     100

  /**
   * register a new MemoryObject
   */
  void register(MemoryObject object);

  /**
   * de-register a MemoryObject. this should only be called by the finaliser,
   * after it is certain the application (including finalisers etc) has
   * completely finished with the object.                         110
   */
  void deRegister(MemoryObject object);

}
```

C.1.4 TypeInterface

```
package openKernel.objectManager;
/**
 * TypeInterface - the connection between the garbage collector and
 * the time-information generated at run-time.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

abstract class TypeInterface {

  /**
   * getOutGoingReferences returns an array of indexes into the
   * Workspace indicating the targets of outwards references in
   * object
   */
  abstract short[] getOutGoingPointers(MemoryObject object);
}
```

C.1.5 ObjectSet

```
package openKernel.objectManager;

/**
 * ObjectSet - an interface to all classes which represent containers of
 * Objects.
 *
 * ObjectSet is little more than a factory interface for Iterators.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 * @see      openKernel.objectManager.Iterator
 */

public interface ObjectSet {

  /**
   * Returns an Iterator over the container.
   */
  Iterator  newIterator();

}
```

C.1.6 Colour

```
package openKernel.objectManager;
/**
 * Colour an ObjectSet with an itegrated Iterator
 *
```

```
 * @version  0.2, June 1997
 * @author   stuart yeates
 * @see      openKernel.objectManager.Iterator
 */

public abstract class Colour implements  ObjectSet, Iterator, AbstractCirclist {
  /**
   * remove a MemoryObject from the container
   */
  abstract void remove(MemoryObject index);

  /**
   * insert a MemoryObject into the container
   */
  abstract void insert(MemoryObject index);

  /**
   * perform a join with another Colour
   */
  abstract void join(Colour colour);
}
```

C.1.7 ReferenceSet

ReferenceSet was implemented as a raw array of shorts, each an index into the array in Workspace.memory().

C.1.8 RootSet

```
package openKernel.objectManager;
/**
 * Interface RootSet, the interface between the RootSet and the garbage
 * collector.
 *
 * Note: that RootSet CANNOT be implemented using a Snaplist, as the fields in
 * MemoryObject used by Snapist are used by the Snaplists in TriColour.
 *
 * Note: the iterated over are not necessarily the roots added and removed
 * from the set. For example, an implementation could add and remove stack
 * bases, but return an iterator over each frame in each stack.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

public interface RootSet extends ObjectSet {

  /**
   * Add a root to the RootSet
   * @param root the root to be added
   */
  void addRoot(MemoryObject root);
```

```
  /**
   * Remove a root from the rootSet
   * @param root the root to be removed
   */
  void removeRoot(MemoryObject root);

  /**
   * Returns an Iterator over the container.
   */
  Iterator  newIterator();

};
```

C.1.9 ExternalRoots

```
package openKernel.objectManager;
/**
 * Interface RootSet, the interface between the RootSet and the garbage
 * collector.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

interface ExternalRoots extends RootSet { }
```

C.2 Implementation Classes

C.2.1 MSMutator

```
package openKernel.objectManager;
/**
 * A Mutator (application) to exercise a garbage collector
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

public class MSMutator extends java.lang.Object {

  static Workspace workspace;
  static MemoryObject[] memory;

  static MutatorNode root;
  static MutatorNode root2;

  static Collector collector;

  /**
   * Create a MutatorNode  and trick the Workspace into thinking that the
   * MutatorNode was created by it not by us
   */
  static MutatorNode fudgeNode(){
```

```
    MemoryObject tmp2;
    MutatorNode tmp;

    tmp = new MutatorNode();

    tmp2 = workspace.get();
    tmp._id = tmp2._id;
    System.out.println(tmp2._id);                                    30
    memory[tmp._id] = tmp;
    collector.register(tmp);
    return tmp;
  }



  public static void main(String[] argv){
    System.runFinalizersOnExit(true);    // tell the VM to run finalisers
                                                                     40
    // initialise the workspace
    workspace = Workspace.instance();
    memory = workspace.memory();
    System.out.println("Workspace initialised         " + workspace);

    // create the Collector;
    collector = MSCollector.getInstance();
    System.out.println("Collector initialised         "+ collector);

    root = fudgeNode();                                              50
    System.out.println("First Mutator Node initialised "+ root);

    collector.addRoot(root);
    System.out.println("First Mutator Node added as a root "+ root);
    {
      collector.writeBarrier(root._id);
      root.left(fudgeNode()._id);
    }
    {
      collector.writeBarrier(root._id);                              60
      root.right(fudgeNode()._id);
    }
    {
      collector.writeBarrier(root.left());
      ((MutatorNode)memory[root.left()]).left(fudgeNode()._id);
      ((MutatorNode)memory[root.left()]).right(fudgeNode()._id);
    }
    {
      collector.writeBarrier(root.right());
      ((MutatorNode)memory[root.right()]).left(fudgeNode()._id);     70
      ((MutatorNode)memory[root.right()]).right(fudgeNode()._id);
    }
    System.out.println("Stack populated");

    fudgeNode();
    fudgeNode();
    fudgeNode();
    fudgeNode();
```

```
System.out.println(collector);
System.out.println("Four MutatorNodes created and set adrift");

Util.show();
collector.completeCollection();
System.out.println("complete collection finished");
System.out.println(collector);
Util.show();
collector.completeCollection();
System.out.println("complete collection finished");
System.out.println(collector);
Util.show();
System.out.println();
System.out.println("beginning second cycle");

root2 = fudgeNode();
collector.addRoot(root2);
{
  collector.writeBarrier(root2._id);
  root2.left(fudgeNode()._id);
}
{
  collector.writeBarrier(root2._id);
  root2.right(fudgeNode()._id);
}
{
  collector.writeBarrier(root2.left());
  ((MutatorNode)memory[root2.left()]).left(fudgeNode()._id);
  ((MutatorNode)memory[root2.left()]).right(fudgeNode()._id);
}
{
  collector.writeBarrier(root2.right());
  ((MutatorNode)memory[root2.right()]).left(fudgeNode()._id);
  ((MutatorNode)memory[root2.right()]).right(fudgeNode()._id);
}
System.out.println("Stack populated");

/* simulate some mutator actions of the heap - if any objects get
   deregistered during this phase we've failed */

int i = 0;
while (i < 100){
  /* create some MutatorNodes and trigger the writeBarrier on some of them */
  fudgeNode().left(Short.MAX_VALUE);
  fudgeNode().left(Short.MAX_VALUE);
  fudgeNode();
  {
    /* rearrange the heap slightly */
    short tmp = ((MutatorNode)memory[root2.left()]).left();
    short tmp2= ((MutatorNode)memory[root2.left()]).right();
    ((MutatorNode)memory[root2.left()]).left(((MutatorNode)memory[root.left()]).right());
    ((MutatorNode)memory[root2.left()]).right(((MutatorNode)memory[root.left()]).left());
    ((MutatorNode)memory[root.left()]).left(tmp);
    ((MutatorNode)memory[root.left()]).right(tmp2);
  }
  i++;
```

```
    }

    Util.show();
    collector.completeCollection();
    System.out.println("complete collection finished");
    System.out.println(collector);
    Util.show();
    System.out.println("Removing first root");
    collector.removeRoot(root);
    collector.completeCollection();
    System.out.println("complete collection finished");
    System.out.println(collector);
    Util.show();

    System.out.println("Removing second root");
    collector.removeRoot(root2);
    collector.completeCollection();
    System.out.println("complete collection finished");
    System.out.println(collector);
    Util.show();
    collector.completeCollection();
    System.out.println("complete collection finished");
    System.out.println(collector);
    Util.show();
  }
}
```

C.2.2 MSCollector

```
package openKernel.objectManager;
/**
 * Mark-and-Sweep Collector, the interface between the Garbage Collector
 * and the rest of the system
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

class MSCollector extends Collector {

  /** The RootSet */
  private RootSet rootSet;

  /** The Algorithm */
  private Algorithm algorithm;

  /** The Workspace memory */
  protected MemoryObject[] _m;

  /**
   * Get the singleton instance of this class
   * @returns the instance
   */
  static public Collector getInstance(){
```

```
  if (collector != null)
    return collector;
  else
    return (collector = new MSCollector());
}

/** The constructor */
private MSCollector() {
  _m = Workspace.instance().memory();
  rootSet = new LinkedRootSet();
  algorithm = new MSAlgorithm(this);
}

/**
 * Register a freshly created heap object.
 * @param object the object to be added
 */
public void register(MemoryObject object){
  algorithm.register(object);
}

/**
 * Notify the collector of a new root (pointer from outside the
 * heap into the heap).
 * @param root the root to be added
 */
public void addRoot(MemoryObject root){
   rootSet.addRoot(root);
   algorithm.newRoot(root);
}

/**
 * Notify the collector of the removal of a root
 * @param root the root to be removed
 */
public void removeRoot(MemoryObject root){
   rootSet.removeRoot(root);
}

/** Get the roots */
final Iterator getRoots(){
  return rootSet.newIterator();
}

/**
 * Perform a small amount of garbage collection
 */
final public void doQuanta(){
  algorithm.doQuanta();
}

/**
 * Perform a large amount of garbage collection
 */
final public void completeCollection(){
  algorithm.completeCollection();
```

```
  }

  /**
   * Maintain the Read-Barrier. Called when the Application is
   * about to read object i. Not required if writeBarrier() is
   * being used.
   */
  final public void readBarrier(short i){
    throw new java.lang.Error();
  }

  /**
   * Maintain the Write-Barrier. Called when the Application is
   * about to write to a field in object i. Not required if
   * readBarrier() is being used.
   */
  final public void writeBarrier(short i){
    algorithm. writeBarrier(i);
  }

  /**
   * Is it necessary to use the barriers at the moment ?
   */
  final public boolean active(){
    return algorithm.active();
  }

  public String toString(){
    return super.toString() + " [ "+ rootSet.toString() + " "+ algorithm.toString() + " ] ";
  }
}
```

C.2.3 MSAlgorithm

```
package openKernel.objectManager;
/**
 * Mark and Sweep Algorithm, the driving algorithm of the garbage collector
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

class MSAlgorithm extends Algorithm{

  /** The Workspace */
  private Workspace workspace;

  /** The Workspace memory */
  protected MemoryObject[] _m;

  /** The Collector */
  private MSCollector collector;

  /** The TriColour */
```

```
private TriColour triColour;

/** The reference iterator - to be reused rather than recycled to avoid
  object creation overhead */
private ReferenceIterator iterator;

/** The number of objects traced since the last flip() */
private int traceCount;

/** The minimum number of objects to trace before a (normal) flip */
private final int traceLimit = 10;

/** The constructor */
MSAlgorithm(MSCollector mSColloector) {
  workspace = Workspace.instance();
  _m =  workspace.memory();
  collector = mSColloector;
  triColour = new TTriColour(1, new MSTriColour());
  iterator = new ReferenceIterator();
}

/**
 * Register a freshly created heap object.
 * @param object the object to be added
 */
public void register(MemoryObject object){
  triColour.register(object);
  doQuanta();
  doQuanta();
}

/**
 * Register a freshly created root
 * @param object  the new root
 */
public void newRoot(MemoryObject object){
  triColour.markGrey(object);
}

 /**
 * Perform a small amount of garbage collection
 */
public void doQuanta(){
  if (triColour.areMoreGrey() || triColour.areMoreUnreach()){
    if (triColour.areMoreGrey())
      trace(triColour.nextGrey());
    if (triColour.areMoreUnreach())
      finish(triColour.nextUnreach());
  } else {
    if (traceCount >= traceLimit){
      triColour.flip(collector.getRoots());
      traceCount = 0;
    }
  }
}
```

```
  /**
   * Perform a large amount of garbage collection
   */
  public void completeCollection(){                                    80
    /* finish the current collection */
    while(triColour.areMoreGrey())
      trace(triColour.nextGrey());

    /* perform a flip */
    triColour.flip(collector.getRoots());
    traceCount = 0;

    /* trace all objects */
    while(triColour.areMoreGrey())                                     90
      trace(triColour.nextGrey());

    /* perform another flip */
    triColour.flip(collector.getRoots());
    traceCount = 0;

    /* finalise all objects */
    while(triColour.areMoreUnreach())
      finish(triColour.nextUnreach());
  }                                                                   100

  /**
   * Maintain the Read-Barrier. Called when the Application is
   * about to read object i. Not required if writeBarrier() is
   * being used. Generally use of readBarrier() is discouraged
   * if use of writeBarrier() is possible.
   */
  final public void readBarrier(short i){
    if (active()){
      if ((i != Short.MAX_VALUE) && (i != Short.MIN_VALUE)){           110
        if (triColour.isGrey(_m[i])||triColour.isWhite(_m[i]))
          trace(_m[i]);
//      System.out.println("readBarrier " + i + " from " + _m[i]);
      } else {
//      System.out.println("readBarrier " + i );
      }
    }
  }

  /**                                                                 120
   * Maintain the Write-Barrier. Called when the Application is
   * about to write to a field in object i. Not required if
   * readBarrier() is being used.
   */
  final public void writeBarrier(short i){
    if (active()){
      if ((i != Short.MAX_VALUE) && (i != Short.MIN_VALUE)){
        if (triColour.isGrey(_m[i]))
          trace(_m[i]);
//      System.out.println("writeBarrier " + i + " from " + _m[i]);   130
      } else {
//      System.out.println("writeBarrier " + i );
```

```
      }
    }
  }

  /**
   * Is it necessary to use the barriers at the moment ?
   */
  final public boolean active(){                                          140
    return true;
  }

  public String toString(){
    return super.toString() + " [ " + triColour.toString() + " ] ";
  }

  /** finish - consider an object for finalisation */
  void finish(MemoryObject object){
    triColour.deRegister(object);                                         150
  }

  /* * * * * * * * * * * * * *  private methods  * * * * * * * * * * * * */

  /**
   * trace all out going references from a MemoryObject
   */
  final private void trace(MemoryObject object){
    short current;
    iterator.initialise(object);                                          160
    traceCount++;
    while(!iterator.isDone()){
      current = iterator.current();
      if ((current != Short.MAX_VALUE) && (current != Short.MIN_VALUE)){
        if (triColour.isWhite(_m[current]))
          triColour.markGrey(_m[current]);
        //      System.out.println("traced " + object + " from " + _m[current]);
      }
      iterator.next();
    }                                                                     170
    triColour.markBlack(object);
  }
}
```

C.2.4 MSTriColour

```
package openKernel.objectManager;
/**
 * An implemenation of TriColour using Snaplists suitable for implementing
 * a Mark-and-Sweep collector.
 *
 * White objects are of unknown reachability.
 *
 * Grey objects are reachable, but have pointers which have not been
 *   examined. they represent the current fringe of the traversal of
 *   the collector through the heap graph.                                10
```

```
 *
 * Black objects are reachable, and finished with.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 * @see      baker78
 */

class MSTriColour implements TriColour{

  /** the four lists of objects */
  private final Colour[] lists = { null, null, null, null };

  private static Workspace workspace;
  private static MemoryObject[] _m;

  /* variables for the four states. these can't be constants, because */
  /* they change every flip  */

  final private byte GREY = 0;
        private byte BLACK = 1;
        private byte WHITE = 2;
  final private byte UNREACH = 3;

  MSTriColour(){
    workspace = Workspace.instance();
    _m = workspace.memory();
    lists[0] = new SnaplistColour();
    lists[1] = new SnaplistColour();
    lists[2] = new SnaplistColour();
    lists[3] = new SnaplistColour();
  }

  /**
   * is this object marked grey ?
   * @return true if object is grey
   * @param object  the object under consideration
   */
  public boolean isGrey(MemoryObject object){return (object._tag == GREY);}


  /**
   * is this object marked white ?
   * @return true if object is white
   * @param object  the object under consideration
   */
  public boolean isWhite(MemoryObject object){return (object._tag == WHITE);}

  /**
   * is this object marked black ?
   * @return true if object is black
   * @param object  the object under consideration
   */
  public boolean isBlack(MemoryObject object){return (object._tag == BLACK);}

   /**
```

```
   * is this object marked black ?
   * @return true if object is black
   * @param object  the object under consideration
   */
  public boolean isMember(MemoryObject object){
    return ((object._tag >=0) && (object._tag <=3));}

//  /**
//   * is this object marked unreachable ?
//   * @return true if object is marked unreachable
//   * @param object  the object under consideration
//   */
//  public boolean isUnreach(MemoryObject object){return (object._tag == UNREACH);}

  /**
   * mark an object grey
   * @param object  the object to be marked
   */
  public void markGrey(MemoryObject object){
    if (isGrey(object)) return ;
    lists[object._tag].remove(object);
    object._tag = GREY;
    lists[GREY].putFirst(object);
  }

  /**
   * mark an object white
   * @param object  the object to be marked
   */
  public void markWhite(MemoryObject object){
    if (isWhite(object)) return ;
    lists[object._tag].remove(object);
    object._tag = WHITE;
    lists[WHITE].putFirst(object);
  }

  /**
   * mark an object black
   * @param object  the object to be marked
   */
  public void markBlack(MemoryObject object){
    if (isBlack(object)) return ;
    lists[object._tag].remove(object);
    object._tag = BLACK;
    lists[BLACK].putFirst(object);
  }

  /** are there more grey objects ? */
  public boolean areMoreGrey(){return !lists[GREY].isDone();}

  /** which is the next grey item ? */
  public MemoryObject nextGrey(){return _m[lists[GREY].first()];}

  /** are there more unreachable objects ? */
  public boolean areMoreUnreach(){return !lists[UNREACH].isDone();}
```

```
  /** which is the next unreachable item ? */
  public MemoryObject nextUnreach(){return _m[lists[UNREACH].first()];}

//  /** get a list of unreachable objects. */
//  public Iterator getUnreachObjects(){
//    return (lists[UNREACH]).newIterator();
//  }
                                                                          130
  /** start a new garbage collection cycle */
  public void flip(Iterator rootSet){

    lists[UNREACH].join(lists[WHITE]);

    /* rotate the list to avoid a list construction and destruction */
    {
      byte tmp = WHITE;
      WHITE = BLACK;
      BLACK = tmp;                                                        140
    }
    /* mark the root grey */
    while (!rootSet.isDone()){
      markGrey(_m[rootSet.current()]);
      rootSet.next();
    }

  }

  /** register a new MemoryObject */                                      150
  public void register(MemoryObject object){
    object._tag = GREY;
    lists[GREY].putFirst(object);
  }

  /**
   * de-register a MemoryObject. this should only be called by the finaliser,
   * after it is certain the application (including finalisers etc) has
   * completely finished with the object.
   */                                                                     160
  public void deRegister(MemoryObject object){
      lists[UNREACH].remove(object);
      object._next = Short.MIN_VALUE;
      object._prev = Short.MIN_VALUE;
      object._tag = Byte.MIN_VALUE;
      workspace.put(object);
  }

  public String toString(){
    return super.toString() + " [  white = "+ lists[WHITE] +" grey = " + lists[GREY] + " black = " + lists[BLA
  }


  public static void main(String[] argv){
    TriColour triColour  = new TTriColour(0, new MSTriColour());
    MemoryObject object = Workspace.instance().get();
    MemoryObject object2 = Workspace.instance().get();
    MemoryObject object3 = Workspace.instance().get();
```

```
    MemoryObject object4 = Workspace.instance().get();
    MemoryObject object5 = Workspace.instance().get();                    180
    object = Workspace.instance().get();
    System.out.println(triColour);
    triColour.register(object);
    System.out.println(triColour);
    triColour.markGrey(object);
    System.out.println(triColour);
    triColour.markBlack(object);
    System.out.println(triColour);
    triColour.markWhite(object);
    System.out.println(triColour);                                        190
    triColour.register(object2);
    triColour.register(object3);
    triColour.register(object4);
    triColour.register(object5);
    System.out.println(triColour);
    triColour.markBlack(object3);
    System.out.println(triColour);
    triColour.markBlack(object5);
    System.out.println(triColour);
    triColour.markBlack(object4);                                         200
    System.out.println(triColour);
    triColour.markBlack(object2);
    System.out.println(triColour);
  }
}
```

C.2.5 GenMutator

```
package openKernel.objectManager;
import java.lang.System;
import java.util.Vector;
import java.util.Enumeration;
/**
 * A Mutator (application) to exercise the garbage collector
 * @version  0.2, June 1997
 * @author   stuart yeates
 */
                                                                          10
public class GenMutator extends java.lang.Object {

  static Workspace workspace;
  static MemoryObject[] memory;

  static MutatorNode root;
  static MutatorNode root2;

  static Collector collector;
                                                                          20
  /**
   * Create a MutatorNode  and trick the Workspace into thinking that the
   * MutatorNode was created by it not by us
   */
```

```
static MutatorNode fudgeNode(){
  MemoryObject tmp2;
  MutatorNode tmp;

  tmp = new MutatorNode();

  tmp2 = workspace.get();
  tmp._id = tmp2._id;
  memory[tmp._id] = tmp;
  collector.register(tmp);
  return tmp;
}

public static void main(String[ ] argv){
  MutatorNode bigRoot;

  System.runFinalizersOnExit(true);    // tell the VM to run finalisers

  // initialise the workspace
  workspace = Workspace.instance();
  memory = workspace.memory();
  System.out.println("Workspace initialised         " + workspace);


  // create the Collector;
  collector = GenCollector.getInstance();
  System.out.println("Generational Collector initialised "+ collector);

  bigRoot = root = fudgeNode();
  collector.addRoot(root);
  {
    collector.writeBarrier(root._id);
    root.left(fudgeNode()._id);
  }
  {
    collector.writeBarrier(root._id);
    root.right(fudgeNode()._id);
  }
  {
    collector.writeBarrier(root.left());
    ((MutatorNode)memory[root.left()]).left(fudgeNode()._id);
    ((MutatorNode)memory[root.left()]).right(fudgeNode()._id);
  }
  {
    collector.writeBarrier(root.right());
    ((MutatorNode)memory[root.right()]).left(fudgeNode()._id);
    ((MutatorNode)memory[root.right()]).right(fudgeNode()._id);
  }
  fudgeNode();
  fudgeNode();
  fudgeNode();
  fudgeNode();
  System.out.println(collector);

  collector.completeCollection();
  System.out.println(collector);
```

```
System.out.println("********* The above collector should have 7 MemoryObjects **********");

collector.completeCollection();
System.out.println(collector);
System.out.println("********* The above collector should have 7 MemoryObjects **********");

root2 = fudgeNode();
collector.addRoot(root2);
{
  collector.writeBarrier(root2._id);                                                    90
  root2.left(fudgeNode()._id);
}
{
  collector.writeBarrier(root2._id);
  root2.right(fudgeNode()._id);
}
{
  collector.writeBarrier(root2.left());
  ((MutatorNode)memory[root2.left()]).left(fudgeNode()._id);
  ((MutatorNode)memory[root2.left()]).right(fudgeNode()._id);                          100
}
{
  collector.writeBarrier(root2.right());
  ((MutatorNode)memory[root2.right()]).left(fudgeNode()._id);
  ((MutatorNode)memory[root2.right()]).right(fudgeNode()._id);
}

collector.completeCollection();
System.out.println(collector);
System.out.println("********* The above collector should have 14 MemoryObjects **********"); 110

/* simulate some mutator actions of the heap - if any objects get
   deregistered during this phase we've failed */

int i = 0;
while (i < 100){
  /* create some MutatorNodes and trigger the writeBarrier on some of them */
  fudgeNode().left(Short.MAX_VALUE);
  fudgeNode().left(Short.MAX_VALUE);
  fudgeNode();                                                                         120
  {
    /* rearrange the heap slightly */
    short tmp = ((MutatorNode)memory[root2.left()]).left();
    short tmp2= ((MutatorNode)memory[root2.left()]).right();
    ((MutatorNode)memory[root2.left()]).left(((MutatorNode)memory[root.left()]).right());
    ((MutatorNode)memory[root2.left()]).right(((MutatorNode)memory[root.left()]).left());
    ((MutatorNode)memory[root.left()]).left(tmp);
    ((MutatorNode)memory[root.left()]).right(tmp2);
  }
  i++;                                                                                 130
}

collector.completeCollection();

collector.removeRoot(root);
collector.completeCollection();
```

```
    collector.removeRoot(root2);
    collector.completeCollection();
                                                                                    140
    collector.removeRoot(bigRoot);
    collector.completeCollection();

    System.out.println(collector);
    System.out.println("********* The above collector should be empty **********");

    testCollector(collector);

  }
  public static void oldmain(String[] argv){                                        150
    // initialise the workspace
    workspace = Workspace.instance();
    memory = workspace.memory();
    System.out.println("Workspace initialised        " + workspace);

    // create the Collector;
    collector = GenCollector.getInstance();
    System.out.println("Collector initialised        "+ collector);

     testCollector(collector);                                                      160

  }

  static void testCollector(Collector collector){
    testRoots(collector);
    testWriteBarriers(collector);
  }
  static void testRoots(Collector collector){

    /* create and add some roots */                                                170
    {
      int i = 0;
      MemoryObject root;

      while(i++ < 10){
        root = fudgeNode();
        collector.addRoot(root);
      }
    }
                                                                                    180
    {
      int i = 0;
      Vector v = new Vector();
      Enumeration e;
      MemoryObject root;

      /* add and remove roots, multiple times ... */
      while(i++ < 20){
        root = fudgeNode();
        collector.addRoot(root);                                                    190
        v.addElement(root);
        System.out.print("1 "+ i+ " ");
```

```
      }

      e = v.elements();
      while(e.hasMoreElements()){
        collector.addRoot((MemoryObject)e.nextElement());
        System.out.print("2");
      }
      System.out.println();                                                    200
      System.out.println(collector);
      System.out.println("********* The above collector should be full **********");
      System.out.println(collector);
      System.out.println("********* The above collector should be full **********");

      e = v.elements();
      while(e.hasMoreElements()){
        collector.removeRoot((MemoryObject)e.nextElement());
        System.out.print("3");
      }                                                                        210

      /* remove items from an empty rootset */
      e = v.elements();
      while(e.hasMoreElements()){
        collector.removeRoot((MemoryObject)e.nextElement());
        System.out.print("4");
      }

    }
    System.out.println();                                                      220
    System.out.println(collector);
    System.out.println(collector);
    System.out.println("********* The above collector should be empty **********");
  }

  /** test the write barriers on the empty collector */
  private static void testWriteBarriers(Collector collector){

    /* test the write barriers on the empty collector */
    short i = -10;                                                             230
    while( i < Workspace.MAX_OBJECTS*2){
      collector.writeBarrier(i++);
      System.out.print("5");
    }

    i = Short.MAX_VALUE;
    while( i > (Short.MAX_VALUE - 100)){
      collector.writeBarrier(i--);
      System.out.print("6");
    }                                                                          240
      i = Short.MIN_VALUE;
      while( i < (Short.MIN_VALUE + 100)){
        collector.writeBarrier(i++);
        System.out.print("7");
      }
    System.out.println();
  }
}
```

C.2.6 GenCollector

```
package openKernel.objectManager;
/**
 * Mark-and-Sweep Collector, the interface between the Garbage Collector
 * and the rest of the system
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

class GenCollector extends Collector {                                    10

  /** The RootSet */
  private RootSet rootSet;

  /** The RootSet Iterator */
  private Iterator rootSetIterator;

  /** The Algorithm */
  private Algorithm algorithm;
                                                                          20
  /** The Workspace memory */
  protected MemoryObject[] _m;

  /**
   * Get the singleton instance of this class
   * @returns the instance
   */
  static public Collector getInstance(){
    if (collector != null)
      return collector;                                                   30
    else
      return (collector = new GenCollector());
  }

  /** The constructor */
  private GenCollector() {
    _m = Workspace.instance().memory();
    rootSet = new LinkedRootSet();
    algorithm = new GenAlgorithm(this);
    rootSetIterator = rootSet.newIterator();                              40
  }

  /**
   * Register a freshly created heap object.
   * @param object the object to be added
   */
  public void register(MemoryObject object){
    algorithm.register(object);
  }
                                                                          50
  /**
```

```
 * Notify the collector of a new root (pointer from outside the
 * heap into the heap).
 * @param root the root to be added
 */
public void addRoot(MemoryObject root){
   rootSet.addRoot(root);
   algorithm.newRoot(root);
}
                                                                    60
/**
 * Notify the collector of the removal of a root
 * @param root the root to be removed
 */
public void removeRoot(MemoryObject root){
   rootSet.removeRoot(root);
}

/** Get the roots */
final Iterator getRoots(){                                          70
  rootSetIterator.reset();
  return rootSetIterator;
}

/**
 * Perform a small amount of garbage collection
 */
final public void doQuanta(){
  algorithm.doQuanta();
}                                                                   80

/**
 * Perform a large amount of garbage collection
 */
final public void completeCollection(){
  algorithm.completeCollection();
}

/**
 * GenCollector is for use only with writeBarrier                   90
 */
final public void readBarrier(short i){
  throw new java.lang.Error("Class GenCollector does not support a " +
                        "readBarrier, use the readBarrier instead.");
}

/**
 * Maintain the Write-Barrier. Called when the Application is
 * about to write to a field in object i. Not required if
 * readBarrier() is being used.                                     100
 */
final public void writeBarrier(short i){
  algorithm. writeBarrier(i);
}

/**
 * Is it necessary to use the barriers at the moment ?
```

```
   */
  final public boolean active(){
    return algorithm.active();
  }

  public String toString(){
    return super.toString() + " [ " +  rootSet.toString() +
      " " + algorithm.toString() + " ] ";
  }
}
```

C.2.7 GenAlgorithm

```
package openKernel.objectManager;
/**
 * Generational Algorithm, the driving algorithm of the garbage collector
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

class GenAlgorithm extends Algorithm{

  /** The Workspace */
  private Workspace workspace;

  /** The Workspace memory */
  protected MemoryObject[] _m;

  /** The Collector */
  private GenCollector collector;

  /** The TriColour holding the older generation */
  private TriColour oldTriColour;

  /** The TriColour holding the youngerer generation */
  private TriColour newTriColour;

  /** The reference iterator - to be reused rather than recycled to avoid
    object creation overhead */
  private ReferenceIterator iterator;

  /** The number of objects traced since the last flip() */
  private int traceCount;


  /** The minimum number of objects to trace before a (normal) flip */
  private final int traceLimit = 10;

  /** The constructor */
  GenAlgorithm(GenCollector genColloector) {
    workspace = Workspace.instance();
    _m = workspace.memory();
    collector = genColloector;
```

```
    newTriColour = new TTriColour(1,new GenTriColour(0));
    oldTriColour = new TTriColour(2,new GenTriColour(1));
//      newTriColour = new GenTriColour(0);
//      oldTriColour = new GenTriColour(1);
    iterator = new ReferenceIterator();
  }

  /**
   * Register a freshly created heap object.                                    50
   * @param object the object to be added
   */
  public void register(MemoryObject object){
    doQuanta();
    doQuanta();
    doQuanta();
    newTriColour.register(object);
  }

  /**                                                                          60
   * Register a freshly created root
   * @param object  the new root
   */
  public void newRoot(MemoryObject object){
    //oldTriColour.markGrey(object);
  }

   /**
   * Perform a small amount of garbage collection
   */                                                                         70
  public void doQuanta(){
    boolean workDone = false;

    /* do a little work on the oldTriColour */
    if (oldTriColour.areMoreGrey()){
      trace(oldTriColour.nextGrey());
      workDone = true;
    }
    if (oldTriColour.areMoreUnreach()){
      finish(oldTriColour.nextUnreach());                                     80
      workDone = true;
    }

    /* do a little work on the newTriColour */
    if (newTriColour.areMoreGrey()){
      trace(newTriColour.nextGrey());
      workDone = true;
    }
    if (newTriColour.areMoreUnreach()){
      finish(newTriColour.nextUnreach());                                     90
      workDone = true;
    }

    if (!workDone)
      if (traceCount >= traceLimit){
        oldTriColour.flip(collector.getRoots());
        traceCount = 0;
```

```
    }

  }

  /**
   * Perform a large amount of garbage collection
   */
  public void completeCollection(){
    /* finish the current collection */
    while(oldTriColour.areMoreGrey())
      trace(oldTriColour.nextGrey());

    /* perform a flip */
    oldTriColour.flip(collector.getRoots());
    traceCount = 0;

    /* trace all objects */
    while(oldTriColour.areMoreGrey())
      trace(oldTriColour.nextGrey());

    /* perform another flip */
    oldTriColour.flip(collector.getRoots());
    traceCount = 0;

    /* finalise all objects */
    while(oldTriColour.areMoreUnreach())
      finish(oldTriColour.nextUnreach());
  }

  /**
   * Maintain the Read-Barrier. Called when the Application is
   * about to read object i. Not required if writeBarrier() is
   * being used.
   */
  final public void readBarrier(short i){
    throw new java.lang.Error("Class GenAlgorithm does not support a " +
                              "readBarrier, use the readBarrier instead.");
  }

  /**
   * Maintain the Write-Barrier. Called when the Application is
   * about to write to a field in object i. Not required if
   * readBarrier() is being used.
   */
  final public void writeBarrier(short i){
    if (active()){
      if ((i >= 0) && (i < Workspace.MAX_OBJECTS)){
        if (oldTriColour.isGrey(_m[i]))
          trace(_m[i]);
//      System.out.println("writeBarrier " + i + " from " + _m[i]);
      } else {
//      System.out.println("writeBarrier " + i );
      }
    }
  }
```

```
  /**
   * Is it necessary to use the barriers at the moment ?
   */
  final public boolean active(){
    return true;
  };

  public String toString(){
    return super.toString() + " [ " + oldTriColour.toString() + " ] ";
  }

  /* * * * * * * * * * * * * * private methods * * * * * * * * * * * * */

  /**
   * trace all out going references from a MemoryObject
   */
  final void trace(MemoryObject object){
    short current;
    iterator.initialise(object);
    traceCount++;
    while(!iterator.isDone()){
      current = iterator.current();
      if ((current != Short.MAX_VALUE) && (current != Short.MIN_VALUE)){
        if (oldTriColour.isWhite(_m[current]))
          oldTriColour.markGrey(_m[current]);
//      System.out.println("traced (true) " + object + " from " + _m[current]);
      } else {
//      System.out.println("traced (false) " + object);
      }
      iterator.next();
    }
    oldTriColour.markBlack(object);
  }

  /** finish - consider an object for finalisation */
  void finish(MemoryObject object){
    oldTriColour.deRegister(object);
  }
}
```

C.2.8 GenTriColour

```
package openKernel.objectManager;
/**
 * An implemenation of TriColour suitable for implementing a two-generation,
 * generational collector.
 *
 * WARNING: There appears to be a bug in this class. the problem stems
 * from the fact there have to be exactly two instances, and they share
 * internal state. The shared state should really be seperated out into
 * another class.
 *
 * flip() has also been rewritten, to ensure that all happens as it should.
 * flip() needs to be called only once for the pair of triColours.
```

```
 *
 * White objects are of unknown reachability.
 *
 * Grey objects are reachable, but have pointers which have not been
 *   examined. they represent the current fringe of the traversal of
 *   the collector through the heap graph.
 *
 * Black objects are reachable, and finished with.                    20
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 * @see      baker78
 */

class GenTriColour implements TriColour{

  /** the nine lists of objects, two sets of four, and one for the set of
   * objects which gets promoted                                      30
   */
  private static final Colour[] lists = { null, null, null, null, null, null,
                                          null, null, null };

  private static Workspace workspace;
  protected static MemoryObject[] _m;

  /** the two triColours */
  static final private int OLD = 0;
  static final private int NEW = 1;                                  40
  static private GenTriColour[] triColours = { null, null };

  /*
   * variables for the four states. these can't be constants, because
   * they change every flip. note that UNREACH is not used as a flag in
   * MemoryObjects becuase they aren't re-coloured when they become
   * unreachable
   */

  private byte GREY;                                                 50
  private byte BLACK;
  private byte WHITE;
  private byte UNREACH;

  /**
   * Construct a TriColour object suitable for use in a generational
   * collector. a pair should be constructed, the first with the argument
   * 0, and the second with 1.
   */
                                                                     60
  GenTriColour(int i){
    /* initialise the workspace */
    workspace = Workspace.instance();
    _m = workspace.memory();

    /* assign the colours */
    switch(i){
    case OLD:
```

```
    triColours[OLD] = this;
    lists[GREY = 0] = new TColour(new SnaplistColour());
    lists[BLACK = 1] = new TColour(new SnaplistColour());
    lists[WHITE = 2] = new TColour(new SnaplistColour());
    lists[UNREACH = 3] = new TColour(new SnaplistColour());
    break;
  case NEW:
    triColours[NEW] = this;
    lists[GREY = 4] = new TColour(new SnaplistColour());
    lists[BLACK = 5] = new TColour(new SnaplistColour());
    lists[WHITE = 6] = new TColour(new SnaplistColour());
    lists[UNREACH = 7] = new TColour(new SnaplistColour());
    break;
  default:
    throw new java.lang.Error("Illegal argument to GenTriColour("+i+")");
  }

  //    System.out.println(this);
}

/**
 * is this object marked grey ?
 * @return true if object is grey
 * @param object  the object under consideration
 */
public boolean isGrey(MemoryObject object){
  return ((object._tag == triColours[OLD].GREY) ||
          (object._tag == triColours[NEW].WHITE));
}


/**
 * is this object marked white ?
 * @return true if object is white
 * @param object  the object under consideration
 */
public boolean isWhite(MemoryObject object){
  return ((object._tag == triColours[OLD].WHITE) ||
          (object._tag == triColours[NEW].WHITE));
}

/**
 * is this object marked black ?
 * @return true if object is black
 * @param object  the object under consideration
 */
public boolean isBlack(MemoryObject object){
  return ((object._tag == triColours[OLD].BLACK) ||
          (object._tag == triColours[NEW].BLACK));
}

/**
 * is this object in this TriColour ?
 * @return true if object is grey
 * @param object  the object under consideration
 */
```

```
final public boolean isMember(MemoryObject object){
  byte b = object._tag;
  if (this == triColours[OLD])
    if ((b >= 0) && (b <=3))
      return true;
    else
      return false;
  else
    if ((b >= 4) && (b <=7))
      return true;
    else
      return false;
}

/**
 * mark an object grey
 * @param object  the object to be marked
 */
public void markGrey(MemoryObject object){
  System.out.println("GenTriColour.markGrey("+object+")");
  Util.show(object);
  if ((isGrey(object)) && isMember(object)) return ;
  if ((object._tag >= 0) && (object._tag < 8))
    lists[object._tag].remove(object);
  else
    System.out.println("doesn't appear to be registered . . .");
  object._tag = GREY;
  lists[GREY].putFirst(object);
  Util.show(object);
}

/**
 * mark an object white
 * @param object  the object to be marked
 */
public void markWhite(MemoryObject object){
  if ((isWhite(object)) && isMember(object)) return ;
  lists[object._tag].remove(object);
  object._tag = WHITE;
  lists[WHITE].putFirst(object);
}

/**
 * mark an object black
 * @param object  the object to be marked
 */
public void markBlack(MemoryObject object){
  if ((isBlack(object)) && isMember(object)) return ;
  lists[object._tag].remove(object);
  object._tag = BLACK;
  lists[BLACK].putFirst(object);
}

/** are there more grey objects ? */
public boolean areMoreGrey(){return !(lists[GREY].isDone());}
```

```
/** which is the next grey item ? */
public MemoryObject nextGrey(){return _m[lists[GREY].first()];}

/** are there more unreachable objects ? */
public boolean areMoreUnreach(){return !(lists[UNREACH].isDone());}

/** which is the next unreachable item ? */
public MemoryObject nextUnreach(){return _m[lists[UNREACH].first()];}

/** start a new garbage collection cycle */
public void flip(Iterator rootSet){
  staticFlip(rootSet);
}

private static void staticFlip(Iterator rootSet){

  /* check for 'problems' */

  /* scehule newly unreachable  objects for destruction */
  lists[triColours[NEW].UNREACH].join(lists[triColours[NEW].WHITE]);
  lists[triColours[OLD].UNREACH].join(lists[triColours[OLD].WHITE]);

  /* promote the NEW objects */
  lists[triColours[OLD].WHITE].join(lists[triColours[NEW].BLACK]);

  /* rotate the list to avoid a list construction and destruction */
  {
    byte tmp = triColours[NEW].WHITE;
    triColours[NEW].WHITE = triColours[NEW].BLACK;
    triColours[NEW].BLACK = tmp;
  }
  {
    byte tmp = triColours[OLD].WHITE;
    triColours[OLD].WHITE = triColours[OLD].BLACK;
    triColours[OLD].BLACK = tmp;
  }

  /* mark the root grey */
  if (rootSet != null)
    while (!rootSet.isDone()){
      triColours[OLD].markGrey(_m[rootSet.current()]);
      rootSet.next();
    }
}

/** register a new MemoryObject */
public void register(MemoryObject object){
  object._tag = GREY;
  lists[GREY].putFirst(object);
}

/**
 * de-register a MemoryObject. this should only be called by the finaliser,
 * after it is certain the application (including finalisers etc) has
 * completely finished with the object.
 */
```

```
  public void deRegister(MemoryObject object){
      lists[UNREACH].remove(object);
      object._next = Short.MIN_VALUE;
      object._prev = Short.MIN_VALUE;                                          240
      object._tag = Byte.MIN_VALUE;
      workspace.put(object);
  }

  public String toString(){
    return super.toString() + " [  "
      + " [ (NEW WHITE) "       + lists[triColours[NEW].WHITE] + " ] "
      + " [ (NEW GREY) "        + lists[triColours[NEW].GREY] + " ] "
      + " [ (NEW BLACK) "       + lists[triColours[NEW].BLACK] + " ] "
      + " [ (NEW UNREACHABLE) " +lists[ triColours[NEW].UNREACH] + " ] "      250
      + " [ (OLD WHITE) "       + lists[triColours[OLD].WHITE] + " ] "
      + " [ (OLD GREY) "        + lists[triColours[OLD].GREY] + " ] "
      + " [ (OLD BLACK) "       + lists[triColours[OLD].BLACK]  + " ] "
      + " [ (OLD UNREACHABLE) " + lists[triColours[OLD].UNREACH] + " ] ";
  }

//   public static void main(String[] argv){
//     GenTriColour triColoura  = new GenTriColour(0);
//     GenTriColour triColour  = new GenTriColour(1);
//     //     MemoryObject[] memory = Workspace.instance().memory();            260
//     MemoryObject object = Workspace.instance().get();
//     MemoryObject object2 = Workspace.instance().get();
//     MemoryObject object3 = Workspace.instance().get();
//     MemoryObject object4 = Workspace.instance().get();
//     MemoryObject object5 = Workspace.instance().get();
//     object = Workspace.instance().get();
//     System.out.println(triColour);
//     triColour.register(object);
//     System.out.println(triColour);
//     triColour.markGrey(object);                                             270
//     System.out.println(triColour);
//     triColour.markBlack(object);
//     System.out.println(triColour);
//     triColour.markWhite(object);
//     System.out.println(triColour);
//     triColour.register(object2);
//     triColour.register(object3);
//     triColour.register(object4);
//     triColour.register(object5);
//     System.out.println(triColour);                                          280
//     triColour.markBlack(object3);
//     System.out.println(triColour);
//     triColour.markBlack(object5);
//     System.out.println(triColour);
//     triColour.markBlack(object4);
//     System.out.println(triColour);
//     triColour.markBlack(object2);
//     System.out.println(triColour);
//   }
}                                                                            290
```

C.2.9 Util

```
package openKernel.objectManager;

/**
 * A few utility debugging methods
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

class Util  {

  static void show(MemoryObject n){
    System.out.println(n +
                       " _next "  + n._next +
                       " _prev "  + n._prev +
                       " _id "    + n._id +
                       " _tag "   + n._tag +
                       " _m[_id] " + Workspace.instance().memory()[n._id]._id);
  }

  static void show(){
    int i = 0;
    while(i < 50){
      show(Workspace.instance().memory()[i++]);
    }
  }
}
```

C.2.10 Snaplist

```
package openKernel.objectManager;
/** Snaplist.java - circular (doubly-linked) list.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 * @see openKernel.objectManager.AbstractCirclist
 */

public class Snaplist implements AbstractCirclist {

  private short _head;
  private short _count;
  protected static MemoryObject[]  _m;

  /** return an iterator over the Snaplist.
   *
   * note that remove() must not be called after an iterator is iterating
   * over the list;
   */
  public Iterator  newIterator() {
    return null;
```

```
  }

// ReColouring has been removed, as it was not necessary, and increased the
// coupling between the Snaplist and the MemoryObject from three to four
// fields.
//
// Recolouring can used to merge arbitary Snaplists and keep membership
// queries correct.
//                                                                        30
//  /** Join two lists and colour the tag  */
//  public void join(Snaplist list,byte tag){
//    join(list);
//    reColour(tag);
//  }
//
//  /** recolour a list */
//    public void reColour(byte tag){
//    if (this.isNotEmpty()){
//                                                                        40
//      /* colour the first */
//      _m[_head]._tag = tag;
//      short head = _m[_head]._prev;
//
//      /* count the rest */
//      while (head != _head){
//      _m[head]._tag = tag;
//      head = _m[head]._prev;
//      System.out.println(".");
//      }                                                                 50
//    }
//  }

  /** join two lists */
  public void join(Snaplist list){
    short count1 = this.count();
    short count2 = list.count();

    if (list._head != Short.MAX_VALUE){
      if (this._head == Short.MAX_VALUE){                               60
        this._head = list._head;
      } else {

        short a, b, c, d;

        a = this._head;
        b = _m[this._head]._next;
        c = list._head;
        d = _m[list._head]._next;
                                                                        70
        _m[a]._next = d;
        _m[b]._prev = c;
        _m[c]._next = b;
        _m[d]._prev = a;

      }
    list._head = Short.MAX_VALUE;
```

```
    }

//      /* check we haven't lost any objects */
//      /* comment out this test in prudction code */
//      if ((this.fcount() != (count1+count2)) ||
//       (this.rcount() != (this._count+list._count))){
//         throw new Error();
//      }
    this._count = (short)(this._count + list._count);
    list._count = 0;
  }


  /** the basic constructor */
  public Snaplist() {
      _count = 0;
      _head = Short.MAX_VALUE;
      _m = Workspace.instance().memory();
  }

//    /** is the list empty ? */
//    final public boolean isNotEmpty() {
//      return (!this.isEmpty());
//    }

  /** is the list empty ? */
  final public boolean isEmpty() {
    return (_head == Short.MAX_VALUE);
  }

  /** how many MemoryObjects are in the list ? */
  final public short count() {
    return _count;
  }

   /** verify, as far as possible, the integrity of this list */
   final public boolean verify() {
     if ((_count == fcount()) && (_count == rcount()))
       return true;
     else {
       return false;
     }
   }

   /** how many MemoryObjects are in the list ? (counting forward)*/
   final private short fcount() {
     short count = 0;
     /* check there's at least one object in the list */
     if (this.isEmpty()) return 0;

     /* count it */
     short head = _m[_head]._prev;
     count++;

     /* count the rest */
```

```
    while (head != _head){
      head = _m[head]._prev;
      count++;
    }
    return count;
  }

/** how many MemoryObjects are in the list ? (reverse counted)*/
final private short rcount() {
  short count = 0;
  /* check there's at least one object in the list */
  if (this.isEmpty()) return 0;

  /* count it */
  short head = _m[_head]._next;
  count++;

  /* count the rest */
  while (head != _head){
    head = _m[head]._next;
    count++;
  }
   return count;
}

/** return the first item on the list */
final public short first() {
  return _head;
}

/** return the last item on the list */
final public short last() {
  return _m[_head]._prev;
}

/** which is the next object after this object ? */
final public short next(short object) {
  return _m[object]._next;
}

/** which is the object before this object ? */
final public short prev(short object) {
  return _m[object]._prev;
}

/** get and remove the first object from the list */
public MemoryObject getFirst() {
  short first = first();
  this.remove(_m[first]);
  return _m[first];
}

/** get and remove the last object from the list */
public MemoryObject getLast() {
  short last = last();
  this.remove(_m[last]);
```

```
  return _m[last];                                                    190
}


/** put an object first in the list */
public void putFirst(MemoryObject  object) {
  insert(object, _head);
  _head = object._id;
}

/** put an object last in the list */                                 200
public void putLast(MemoryObject  object) {
  insert(object, _head);
}

private void insert(MemoryObject  object, short head){
  if ((object._next != Short.MAX_VALUE) ||
      (object._prev != Short.MAX_VALUE)){
    System.out.println("Error --- trying to add an object already in a list");
    Thread.dumpStack();
  }                                                                   210
  if (head == Short.MAX_VALUE){
    object._next = object._id;
    object._prev = object._id;
    _head = object._id;
  } else {
    object._next = head;
    object._prev = _m[head]._prev;
    _m[_m[head]._prev]._next = object._id;
    _m[head]._prev = object._id;
  }                                                                   220
  _count++;
}

/** remove a given MemoryObject for the list */
public void remove(MemoryObject  object){

  //    System.out.println("Removing object " + object._id + " from this " + this);
  if ((object._next == Short.MAX_VALUE) ||
      (object._prev == Short.MAX_VALUE)){
    System.out.println("Error --- object not in a list");             230
    Thread.dumpStack();
  }
  if (_head == Short.MAX_VALUE){
    System.out.println("Error --- removing object from an empty list");
    Util.show(object);
    Thread.dumpStack();
  }
  if (_head == object._id){
    if (object._id == object._next) // there's only one item on the list
      _head = Short.MAX_VALUE;                                        240
    else   // there's several items, object is the head
      _head = object._next;
  }

  _m[object._next]._prev = object._prev;
```

```
    _m[object._prev]._next = object._next;

    object._next = Short.MAX_VALUE;
    object._prev = Short.MAX_VALUE;

    _count--;
  }

  /** convert the list to a string */
  public String toString() {
    int count = count();
    short head = _head;
    String s;

    s = /*super.toString() +*/ " (verify="+verify()+
      ", head=" + _head + ", count="+ count() + ") [ " ;
    if (_head != Short.MAX_VALUE)
      while (count-- > 0){
        s = s + _m[head] + " ";
        head = _m[head]._prev;
      }
    s = s + "]";
    return s;
  }

  public static void main(String argv[]){

    int counter = 0;
    Snaplist snaplist = new Snaplist();

    System.out.println(snaplist);
    System.out.println();

    while(counter < 20){
      System.out.println(_m[counter]);
      snaplist.putFirst(_m[counter++]);
      System.out.println(snaplist);
      System.out.println(_m[counter]);
      snaplist.putLast(_m[counter++]);
      System.out.println(snaplist);
      System.out.println();
      Util.show();
    }

    counter = 0;
    while(counter < 20){
      snaplist.getFirst();
      snaplist.remove(_m[counter]);
      counter += 5;
      System.out.println(snaplist);
      System.out.println();
    }
    System.out.println();
    System.out.println();
    System.out.println();
```

```
    Snaplist snaplist2 = new Snaplist();
    System.out.println(snaplist);
    System.out.println(snaplist2);
    Util.show();
    System.out.println();

    snaplist.join(snaplist2);
    System.out.println(snaplist);
    System.out.println(snaplist2);                                          310
    Util.show();
    System.out.println();

    snaplist2.join(snaplist);
    System.out.println(snaplist);
    System.out.println(snaplist2);
    Util.show();

    System.out.println();
    System.out.println();                                                   320
    System.out.println();
    System.out.println();

    counter = 25;
    while(counter < 45){
      snaplist.putFirst(_m[counter++]);
      snaplist.putLast(_m[counter++]);
//       snaplist.reColour((byte) counter);
    }
    System.out.println(snaplist);                                           330
    System.out.println(snaplist2);
    Util.show();

    System.out.println();
    System.out.println();
    System.out.println();
    System.out.println();

    snaplist2.join(snaplist);
    System.out.println(snaplist);                                           340
    System.out.println(snaplist2);
    Util.show();
    System.out.println(snaplist2.count());
    //     System.out.println(snaplist2.rcount());
    System.out.println(snaplist2);
  }

}
```

C.2.11 SnaplistColour

```
package openKernel.objectManager;
/*
 * SnaplistColour - an implementation of Colour using Snaplists.
 *
```

```
 * Uses a depth-first traversal mechanism.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 * @see openKernel.objectManager.Colour
 * @see openKernel.objectManager.Iterator                                    10
 * @see openKernel.objectManager.AbstractCirclist
 */

final public class SnaplistColour extends Colour {

  Snaplist snaplist = null;

  SnaplistColour(){
    snaplist = new Snaplist();
  }                                                                          20

  /* Colour methods */

  final public void remove(MemoryObject index){snaplist.remove(index);}
  final public void insert(MemoryObject index){this.putFirst(index);  }

  /* ObjectSet methods */

  final public Iterator  newIterator(){return this;}
                                                                             30
  /* Iterator methods */

  final public void next(){}
  final public void reset(){}
  final public short current(){return snaplist.first();}
  final public boolean isDone(){return snaplist.isEmpty();}

  /* AbstractCirclist methods */

  final public void        putFirst(MemoryObject index){snaplist.putFirst(index);}   40
  final public void        putLast(MemoryObject index) {snaplist.putLast(index);}
  final public MemoryObject getFirst()                  {return snaplist.getFirst();}
  final public short       count()                    {return snaplist.count();}
  final public short       first()                   {return snaplist.first();}
  final public short       next(short index){          return snaplist.next(index);}

  final public void join(SnaplistColour snaplistColour){
    snaplist.join(snaplistColour.snaplist);
  }
  final public void join(Colour colour){                                     50
    snaplist.join(((SnaplistColour)colour).snaplist);
  }

  public String toString(){
    return snaplist.toString();
  }

}
```

C.2.12 RSnaplist

```
package openKernel.objectManager;
/**
 * RSnaplist.java - a circular (doubly-linked) list that may be Robustly
 * iterated over.
 *
 * Works by registering all iterators created and notifying them of
 * operations on the list which may effect them.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates                                              10
 */

public class RSnaplist extends Snaplist {

  /** the array of iterators of the list */
  private RIterator[] iterators;

  /** the maximum number of iterators over the list */
  private final static int MAX_ITERATORS = 5;
                                                                        20
  /** return an iterator over the RSnaplist.  */
  public Iterator newIterator() {

    // create the iterator
    RIterator iterator = new NormalRSnaplistIterator(this);

    // add the iterator to the list
    this.addIterator(iterator);

    // return it;                                                       30
    return iterator;
  }

  /** return an iterator over the RSnaplist.  */
  public Iterator newRobustIterator() {

    // create the iterator
    RIterator iterator = new CompleteRSnaplistIterator(this);

    // add the iterator to the list                                     40
    this.addIterator(iterator);

    // return it;
    return iterator;
  }

  /** remove this iterator from the list of active iterators */
  void removeIterator(RIterator iterator){
    int i = 0;
    while ((i < MAX_ITERATORS) && (iterators[i] != iterator))           50
      i++;
    if (iterators[i] == iterator) iterators[i] = null;
  }
```

```
/** add this iterator from the list of active iterators */
void addIterator(RIterator iterator){
  int i = 0;
  while ((i < MAX_ITERATORS) && (iterators[i] != null))
    i++;
  if (i == MAX_ITERATORS) throw new java.lang.Error();                    60
  iterators[i] = iterator;
}

/** the basic constructor */
public  RSnaplist() {
  super();
  iterators = new RIterator[MAX_ITERATORS];
}

/** put an object first in the list */                                     70
final public void putFirst(MemoryObject  aLink) {
  int i = 0;
  while (i < MAX_ITERATORS){
    if (iterators[i] != null)
      iterators[i].notifyInsertFirst(aLink._id);
    i++;
  }
  super.putFirst(aLink);
}
                                                                         80
/**put an object last in the list */
final public void putLast(MemoryObject  aLink) {
  int i = 0;
  while (i < MAX_ITERATORS){
    if (iterators[i] != null)
      iterators[i].notifyInsertLast(aLink._id);
    i++;
  }
  super.putLast(aLink);
}                                                                        90

/**remove - remove a given MemoryObject for the list */
final public void remove(MemoryObject  aLink){
  int i = 0;
  while (i < MAX_ITERATORS){
    if (iterators[i] != null)
      iterators[i].notifyDeletion(aLink._id);
    i++;
  }
  super.remove(aLink);                                                   100
}

/** convert the list to a string */
public String toString() {
  short  current = first();
  short  count = count();
  String s;

  s =  "RSnaplist@ [";
```

```
                                                                    110
  s = s + "(" + count + ") ";

  while (count-- > 0){
    s = s + _m[current].toString() + " ";
    current = _m[ current ]._next;
  };

  s = s + "] ";
  return s;
}                                                                   120

/**
 * StubRSnaplistIterator - a stub class to pool the common logic
 * of all robust CircList iterators.
 *
 * It is expected that sub-types will override one or more of the notify
 * method calls. To be truely robust, at least notifyDeletion needs to be
 * overridden.
 *
 */                                                                 130

public class StubRSnaplistIterator extends CirclistIterator implements RIterator {

  /** the container we're iterating over */
  RSnaplist _list;

  /** Creates the iterator by taking a circular list to traverse. */
  public StubRSnaplistIterator(RSnaplist list) {
    super(list);
    _list = list;                                                   140
    _finished= false;
  }

  /** notify the iterator of imminant deletion of an object */
  public void notifyDeletion(short object){ }

  /** notify the iterator of imminant insertion of an object */
  public void notifyInsertionAfter(short old, short inserted){ };

  /** notify the iterator of imminant insertion of an object */      150
  public void notifyInsertFirst(short object){ }

  /** notify the iterator of imminant insertion of an object */
  public void notifyInsertLast(short object){ }

  /** Positions the iterator to the next object in the sequence. */
  final public void   next() {
    if (!super.isDone()){
      super.next();
    }                                                               160
  }

  /** Positions the iterator to the first object in the list. */
  final public void reset() {
    super.reset();
```

```
    if (_finished)
      _list.addIterator(this);
  }

  /** has the iterator run out of objects ? */
  final public boolean   isDone() {
    if (!_finished){
      if (super.isDone()){
        _list.removeIterator(this);
        _finished = true;
        return true;
      } else {
        return false;
      }
    } else {
      return true;
    }
  }

  /**
   * if _finished is TRUE, then the iterator has is not only finished,
   * it's been removed from the CircList's list of iterators.
   */
  private  boolean           _finished;
}

/**
 * NormalRSnaplistIterator - an iterator which notices and handles the
 * problems ariseing from the deletion of items from a list.
 *
 * @see openKernel.objectManager.StubRSnaplistIterator
 */
public class NormalRSnaplistIterator extends StubRSnaplistIterator {

  /**
   * Propogate the constructor
   */
  public NormalRSnaplistIterator(RSnaplist list){
    super(list);
  }

  /**
   * If the object being deleted is the current one, advance the iterator
   */
  public void notifyDeletion(short object){
    if (object == this.current()){
      this.next();
    }
  }
}

/**
 * CompleteRSnaplistIterator - an iterator which notices and handles the
 * problems ariseing from the deletion of items from a list. It also
 * detects insertion of objects which may not be iterated over and rewinds
 * the iterator so all objects are iterated over.
```

```
 *
 */

public class CompleteRSnaplistIterator extends StubRSnaplistIterator {

  /**
   * Propogate the constructor
   */
  public CompleteRSnaplistIterator(RSnaplist list){                    230
    super(list);
  }

  /**
   * If the object being deleted is the current one, advance the iterator
   */
  public void notifyDeletion(short object){
    if (object == this.current()){
      this.next();
    }                                                                  240
  }

  /**
   * If the object is being inserted at an unknown position, rewind
   * to ensure we catch it
   */
  public void notifyInsertionAfter(short old, short inserted){
    if (old != this.current())
      this.reset();
  }                                                                    250

  /**
   * Rewind the iterator to ensure the new object is iteratored over
   */
  public void notifyInsertFirst(short object){
    this.reset();
  }

}
                                                                       260
public static void main(String argv[]){

  int counter = 0;
  RSnaplist rSnaplist = new RSnaplist();

  System.out.println(rSnaplist);
  System.out.println();

  while(counter < 50){
    rSnaplist.putFirst(_m[counter++]);                                 270
    rSnaplist.putLast(_m[counter++]);
  }
  System.out.println(rSnaplist);
  System.out.println();

  counter = 0;
  while(counter < 50){
```

```
      rSnaplist.getFirst();
      rSnaplist.remove(_m[counter]);
      counter += 5;                                                    280
    }
     System.out.println(rSnaplist);
       System.out.println();

  }

}
```

C.2.13 TTriColour

```
package openKernel.objectManager;
/**
 * A simple wrapper around a TriColour to print debugging information
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

class TTriColour implements TriColour{
  TriColour triColour;                                                 10
  int number;

  TTriColour(int i, TriColour t){
    triColour = t;
    number = i;
    System.out.println(number + " TTriColour constructor . . .");
  }
  public boolean isGrey(MemoryObject object){
    System.out.println(number + " TTriColour.isGrey(" + object + ")");
    return triColour.isGrey(object);                                   20
  }
  public boolean isWhite(MemoryObject object){
    System.out.println(number + " TTriColour.isWhite(" + object + ")");
    return triColour.isWhite(object);
  }
  public boolean isBlack(MemoryObject object){
    System.out.println(number + " TTriColour.isBlack(" + object + ")");
    return triColour.isBlack(object);
  }
  public boolean isMember(MemoryObject object){                        30
    System.out.println(number + " TTriColour.isMember(" + object + ")");
    return triColour.isMember(object);
  }
  public void markWhite(MemoryObject object){
    System.out.println(number + " TTriColour.markWhite(" + object + ")");
    triColour.markWhite(object);
  }
  public void markGrey(MemoryObject object){
    System.out.println(number + " TTriColour.markGrey(" + object + ")");
    triColour.markGrey(object);                                        40
  }
```

```
  public void markBlack(MemoryObject object){
    System.out.println(number + " TTriColour.markBlack(" + object + ")");
    triColour.markBlack(object);
  }
  public boolean areMoreGrey(){
    boolean t;
    System.out.println(number + " TTriColour.areMoreGrey() = "
                       + (t = triColour.areMoreGrey()));
    return t;                                                              50
  }
  public MemoryObject nextGrey(){
    MemoryObject n;
    System.out.println(number + " TTriColour.nextGrey() = "
                       + (n = triColour. nextGrey()));
    return n;
  }
  public boolean areMoreUnreach(){
    boolean n;
    System.out.println(number + " TTriColour.areMoreUnreach() = " +        60
                       (n = triColour.areMoreUnreach()));
    return n;
  }
  public MemoryObject nextUnreach(){
    MemoryObject n;
    System.out.println(number + " TTriColour.nextUnreach() = " +
                       (n = triColour.nextUnreach()));
    return n;
  }
//   public Iterator getUnreachObjects(){                                  70
//     Iterator i;
//     System.out.println(number + " TTriColour.getUnreachObjects() = " +
//                        (i = triColour.getUnreachObjects()));
//     return i;
//   }
  public void flip(Iterator rootSet){
    System.out.println(number + " TTriColour.flip(rootSet)");
    System.out.println(triColour);
    triColour.flip(rootSet);
    System.out.println(triColour);                                         80
  }
  public void register(MemoryObject object){
    System.out.println(number + " TTriColour.register(" + object + ")");
    triColour.register(object);
  }
  public void deRegister(MemoryObject object){
    System.out.println(number + " TTriColour.deRegister(" + object + ")");
    triColour.deRegister(object);
  }
                                                                           90
  public String toString(){
    return triColour.toString();
  }
}
```

C.2.14 LinkedList

```
package openKernel.objectManager;
/**
 * LinkedList - a linked list class using inner classes.
 *
 * LinkedList is fully self-contained, no assumptions are made about the
 * Object dealt with, the Objects are never accessed, nulls can safely be
 * used in the list.
 *
 * The Iterators and getIterator functions, however, are tied to
 * openKernel.objectManager.MemoryObject and Iterator.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

public class LinkedList {

  /**
   * Node - the inner class representing a node on the list. all fields
   * are wide-open for efficient use by the LinkedList
   */
  private class Node {
    public Node prev;
    public Node next;
    public Object object;

    private Node(){
    }

    /**
     * put - the only method of passing interest - nulls all pointers
     * and adds it to the stack of unused Nodes
     */
    final void put(){
      this.next = unused;
      if (this.next != null)
        this.next.prev = this;
      this.prev = null;
      this.object = null;
      unused = this;
    }
    protected void finalize(){ /* nothing */}
  }

  final private Node getNode(Object object){
    Node tmp;

    if (unused == null){
      /* create a new one if necessary */
      tmp = new Node();
    } else {
      /* return a used one */
      tmp = unused;
      unused = unused.next;
```

```
    }
    tmp.prev = null;
    tmp.next = null;
    tmp.object = object;
    return tmp;
  }                                                                    60
  private Node unused;


  /**
   * Iterate forward through a LinkedList. Assumes that there are no nulls in the
   * list
   */

  class LinkedListForwardIterator implements Iterator {
                                                                       70
    private LinkedList list;
    private Node current;

    public LinkedListForwardIterator(LinkedList list){
      this.list = list;
      this.reset();
    }

    final public void reset(){
      if (current != null)                                             80
        current =  list.head;
    }

    final public void  next(){
      current =  current.next;
    }

    final public short current(){
      return ((MemoryObject)current.object)._id;
    }                                                                  90

    final public boolean isDone(){
      return (current == null);
    }
  }

  /**
   * Iterate backwards through a LinkedList. Assumes that there are no nulls in the
   * list. NOT robust.
   */                                                                 100

  class LinkedListBackwardIterator implements Iterator {

    private LinkedList list;
    private Node current;

    public LinkedListBackwardIterator(LinkedList list){
      this.list = list;
      this.reset();
    }                                                                  110
```

```
  final public void reset(){
    current =  list.tail;
  }

  final public void  next(){
    if (current != null)
      current = current.prev;
  }
                                                                          120
  final public short current(){
    return ((MemoryObject)current.object)._id;
  }

  final public boolean isDone(){
    return (current == null);
  }
}

/**                                                                       130
 * the head of the list
 */
private Node head;

/**
 * the tail of the list
 */
private Node tail;

/**                                                                       140
 * the size of the list;
 */
private int count;

/**
 * the constructor
 */
public LinkedList(){
  head = null;
  tail = null;                                                            150
  count = 0;
}

/**
 * return the size of the list
 */
public int size(){return count;};

/**
 * overrides java.lang.Object.toString() to output more useful information  160
 * @see java.lang.Object.toString()
 */
final public String toString(){
  return super.toString() +  " [" + reString(head) + " ]";
}
```

```
/**
 * a helper function for toString()
 */
final private String reString(Node node){
  if (node!=null)
    return " " + node.object +
      reString(node.prev);
  else
    return "";
}

/**
 * put an object on the head of the list
 * @param  object the object to be added
 * @see java.lang.Object
 */
final public void putFirst(Object object){
  Node tmp = getNode(object);
  if (head==null){
    if (tail!=null) throw new java.lang.Error();
    head = tmp;
    tail = tmp;
    tmp.prev = null;
    tmp.next = null;
  } else {
    if (tail==null) throw new java.lang.Error();
    head.next = tmp;
    tmp.prev = head;
    tmp.next = null;
    head = tmp;
  }
  count++;
}

/**
 * put an object on the tail of the list
 * @param  object the object to be added
 * @see java.lang.Object
 */
final public void putLast(Object object){
  Node tmp = getNode(object);
  if (tail==null){
    if (head!=null) throw new java.lang.Error();
    tail = tmp;
    head = tmp;
    tmp.prev = null;
    tmp.next = null;
  } else {
    if (head==null) throw new java.lang.Error();
    tail.prev = tmp;
    tmp.next = tail;
    tmp.prev = null;
    tail = tmp;
  }
  count++;
}
```

```
/**
 * delete an object from the list
 * @param  object the object to be deleted
 * @see java.lang.Object
 */
final public void delete(Object object){
  Node node = head;                                                    230

  // handle the extreme case
  if (head == last())
    getLast();

  // handle the normal case
  while (node != null){
    if (node.object == object){
      // handle the upstream
      if (node.prev == null){                                          240
        tail = node.next;
      } else {
        node.prev.next = node.next;
      }
      // handle the downstream
      if (node.next == null){
        head = node.prev;
      } else {
        node.next.prev = node.prev;
      }                                                                250
      count--;
      node.put();
      return;
    } else {
      node = node.prev;
    }
  }
}

/**                                                                    260
 * return (without removing) the first object on the list
 * @return the object first on the list
 * @see java.lang.Object
 */
final public Object first(){
  if (head == null)
    return null;
  else
    return head.object;
}                                                                      270

/**
 * return (without removing) the last object on the list
 * @return the object last on the list
 * @see java.lang.Object
 */
final public Object last(){
  if (head == null)
```

```
    return null;
  else
    return tail.object;
}

/**
 * get and remove the first object on the list
 * @return the object formerly first on the list
 * @see java.lang.Object
 */
final public  Object getFirst(){
  Object tmp = first();
  this.delete(tmp);
  return tmp;
}

/**
 * get and remove the last object on the list
 * @return the object formerly last on the list
 * @see java.lang.Object
 */
final public Object getLast(){
  if (head == null){
    return null;
  }
  Node node = head.prev;
  Object object = node.object;

  // handle the upstream
  if (node.prev == null){
    tail = node.next;
  } else {
    node.prev.next = node.next;
  }
  // handle the downstream
  if (node.next == null){
    head = node.prev;
  } else {
    node.next.prev = node.prev;
  }
  count--;
  node.put();
  return object;
}

/**
 * get the previous object in the list
 * @param  object  the 'current' object
 * @return the 'previous' object
 */
final public Object prev(Object object){
  Node node = head;
  while (node != null){
    if (node.object == object){
      if (node.prev == null)
        return null;
```

```
      else
        return node.prev.object;
    } else {
      node = node.prev;
    }
  }
  return null;
}

/**
 * get the next object in the list
 * @param  object  the 'current' object
 * @return the 'next' object
 */
final public Object next(Object object){
  Node node = head;
  while (node != null){
    if (node.object == object){
      return node.next.object;
    } else {
      node = node.next;
    }
  }
  return null;
}

/**
 * Return an iterator over the list
 */
final public Iterator newIterator(){
  return new LinkedListForwardIterator(this);
}

/**
 * Return an iterator over the list
 */
final public Iterator newBackwardIterator(){
  return new LinkedListBackwardIterator(this);
}

/**
 * Return an iterator over the list
 */
final public Iterator newForwardIterator(){
  return new LinkedListForwardIterator(this);
}

/**
 * main - a small function to test the functionality of several key
 * primitive operations on a LinkedList
 */

final static public void main(String argv[]){

  // objects to play with
  Integer a = new Integer(1);
```

```
Integer b = new Integer(2);
Integer c = new Integer(3);
Integer d = new Integer(4);
LinkedList l = new LinkedList();

// run the speedtest
l.speedTest(50);
l.speedTest(100);
l.speedTest(500);
l.speedTest(1000);

// add the integers to the list
System.out.println(l);
l.putFirst(a);
System.out.println(l);
l.putFirst(b);
System.out.println(l);
l.putLast(c);
System.out.println(l);
l.putLast(d);
System.out.println(l);

// throw in a null, to ensure we're reobust
l.putLast(null);
System.out.println();
System.out.println(l);

// a little stack in the unused queue
l.putLast(new Integer(100));
l.putLast(new Integer(101));
l.putLast(new Integer(102));
l.getLast();
l.getLast();
l.getLast();

// Rotate the list a couple of times to exercise the primitives
System.out.println();
System.out.println("Rotating the list ... the memory addresses and numbers " +
                   "should remain identical ... their order should rotate");
l.putLast(l.getFirst());
System.out.println(l);
l.putLast(l.getFirst());
System.out.println(l);
l.putLast(l.getFirst());
System.out.println(l);
l.putLast(l.getFirst());
System.out.println(l);
l.putLast(l.getFirst());
System.out.println();
System.out.println(l);

System.out.println();
System.out.println("Reverse rotating the list ... the memory addresses and " +
                   "numbers should remain identical ... intermediate stages " +
                   "not shown");
l.putFirst(l.getLast());
```

```
    l.putFirst(l.getLast());
    l.putFirst(l.getLast());
    l.putFirst(l.getLast());
    l.putFirst(l.getLast());
    System.out.println();
    System.out.println(l);

    /*
     * delete the integers from the list, exercising all four paths
     * through the delete() method.
     */
    System.out.println();
    System.out.println("Deleting objects in order of insertion . . .");
    l.delete(null);
    System.out.println(l);
    l.delete(a);
    System.out.println(l);
    l.delete(d);
    System.out.println(l);
    l.delete(b);
    System.out.println(l);
    l.delete(c);
    System.out.println(l);
  }
  /* * * * * * * * * * * * * * private methods * * * * * * * * * * * * */
  private void speedTest(int items){
    final int samples = 5;
    final int loops = 1;
    int repeat = 0;
    int count = 0;
    int k = 0;
    long first;
    long second;
    long best = Long.MAX_VALUE;
    Integer[] Ints;

    Ints = new Integer[items+2];
    k = 0;
    while (k++ < items){
      Ints[k] = new Integer(k);
    }

    best = Long.MAX_VALUE;
    repeat = 0;
    while (repeat++ < samples){
      count = 0;
      first = System.currentTimeMillis();
      while (count++ < loops){
        k = 0;
        while (k < items){
          this.putLast(Ints[k]);
          this.delete(Ints[k]);
          k++;
        }
      }
      second = System.currentTimeMillis();
```

```
  if (best > (second - first))
    best = second - first;
}
System.out.println("fastest " + items +
                   " putLast()/remove() matched pairs = " + best);

best = Long.MAX_VALUE;
repeat = 0;
while (repeat++ < samples){
  count = 0;
  first = System.currentTimeMillis();
  while (count++ < loops){
    k = 0;
    while (k < items){
      this.putFirst(Ints[k]);
      this.delete(Ints[k]);
      k++;
    }
  }
  second = System.currentTimeMillis();
  if (best > (second - first))
    best = second - first;
}
System.out.println("fastest " + items +
                   " putFirst()/remove() matched pairs = " + best);

best = Long.MAX_VALUE;
repeat = 0;
while (repeat++ < samples){
  count = 0;
  first = System.currentTimeMillis();
  while (count++ < loops){
    k = 0;
    while (k < items){
      this.putLast(Ints[k++]);
    }
    k = 0;
      while (k < items){
        this.delete(Ints[k++]);
      }
  }
  second = System.currentTimeMillis();
  if (best > (second - first))
    best = second - first;
}
System.out.println("fastest " + items +
                   " putLast()/remove() seperated pairs = " + best);

best = Long.MAX_VALUE;
repeat = 0;
while (repeat++ < samples){
  count = 0;
  first = System.currentTimeMillis();
  while (count++ < loops){
    k = 0;
    while (k < items){
```

```
        this.putFirst(Ints[k++]);
      }
      k = 0;
      while (k < items){
        this.delete(Ints[k++]);
      }
    }
    second = System.currentTimeMillis();
    if (best > (second - first))
      best = second - first;
  }
  System.out.println("fastest " + items +
                     " putFirst()/remove() seperated pairs = " + best);


  best = Long.MAX_VALUE;
  repeat = 0;
  while (repeat++ < samples){
    count = 0;
    first = System.currentTimeMillis();
    while (count++ < loops){
      k = 0;
      while (k++ < items){
        this.putLast(Ints[k]);
      }
      k = 0;
      while (k < items){
        this.delete(Ints[items-k++]);
      }
    }
    second = System.currentTimeMillis();
    if (best > (second - first))
      best = second - first;
  }
  System.out.println("fastest " + items +
                     " putLast()/remove() reversed seperated pairs = " + best);

  best = Long.MAX_VALUE;
  repeat = 0;
  while (repeat++ < samples){
    count = 0;
    first = System.currentTimeMillis();
    while (count++ < loops){
      k = 0;
      while (k++ < items){
        this.putFirst(Ints[k]);
      }
      k = 0;
      while (k < items){
        this.delete(Ints[items-k++]);
      }
    }
    second = System.currentTimeMillis();
    if (best > (second - first))
      best = second - first;
  }
```

```
    System.out.println("fastest "+items+" putFirst()/remove() reversed " +
                       "seperated pairs = "+ best);

    System.out.println("time shown is fastest of "+samples+" samples, times " +
                       "in milli-seconds");
    System.out.println("\"matched pairs\" indicates each item is removed before " +
                       "the next is added");
    System.out.println("\"seperated pairs\" indicates all items are added before " +
                       "any are removed");
    System.out.println("\"reversed seperated pairs\" indicates items are removed " +
                       "in reverse order");
  }
}
```

C.2.15 LinkedRootSet

```
package openKernel.objectManager;
/**
 * An implementation of RootSet using a LinkedList. This implementation is
 * built around the assumption that there is only a single stack, and that
 * the RootSet operates in a stack-like manner. If it is to be used in a
 * multithreaded environment, removing the roots which correspond to "old"
 * stack frames will be inefficient. Should this situation arise, and
 * real-time performance be required, an alternative implementation of RootSet
 * will have to be sought, possible multiple lists, one for each stack, with a
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

class LinkedRootSet implements ExternalRoots {

  /** The linked list holding the roots */
  private LinkedList _list;

  /** The Iterator over the list */
  private Iterator iterator;

  /**
   * The upper bound on the number of roots. This is a soft limit, exceeding it
   * will not cause incorrect results, merely forces the creation of new objects
   * after initialisation.
   */
  private final static int MAX_ROOTS = 200;

  LinkedRootSet(){
    _list = new LinkedList();

    {
      int i = 0;
      while (i++ <= this.MAX_ROOTS)
        _list.putFirst(this);
      while (i-- >= 0)
        _list.delete(this);
```

```
    }
    iterator = _list.newIterator();
  }

  /**
   * Add a root to the RootSet
   * @param root the root to be added
   */
  public void addRoot(MemoryObject root){
    _list.putFirst(root);
  }

  /**
   * Remove a root from the rootSet
   * @param root the root to be removed
   */
  public void removeRoot(MemoryObject root){
    _list.delete(root);
  }

  /**
   * Get an iterator over all current roots
   * @returns Iterator  an iterator over the RootSet
   */
  public Iterator newIterator(){
    iterator.reset();
    return iterator;
  }

  public String toString(){
    return _list.toString();
  }
}
```

C.2.16 MutatorNode

```
package openKernel.objectManager;
import java.lang.Short;
/**
 * MutatorNode - 'Node' class after the attention on the code generator
 * to include calls to the write barrier.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

final public class MutatorNode extends MemoryObject {

  private static Collector collector;

  MutatorNode(){
    if (collector == null)
      collector = Collector.getInstance();
  }
```

```
  /**
   * the array of references in the node
   */
  private short[] references = {Short.MAX_VALUE, Short.MAX_VALUE};

  /**
   * extract the references from the node, over riding the method
   * in MemoryObject
   */
  public final short[] getReferences(){return references;}

  /**
   * examine the value of the 'left' field
   */
  public final short left(){return references[0];}

  /**
   * examine the value of the 'right' field
   */
  public final short right(){return references[1];}

  /**
   * set the value of the 'left' field, preserving the writeBarrier
   */
  public final void left(short s){
    collector.writeBarrier(this._id);
    references[0] = s;
  }

  /**
   * set the value of the 'right' field, preserving the writeBarrier
   */
  public final void right(short s){
    collector.writeBarrier(this._id);
    references[1] = s;
  }

  public final boolean areIdentical(){
    if (this.left() == this.right()){
      this.right(Short.MAX_VALUE);
      return true;
    } else {
      return false;
    }
  }
}
```

C.2.17 Node

```
package openKernel.objectManager;
import java.lang.Short;
/**
 * Node - a simple example node in the heap of an application. There
```

```
 * are two outgoing pointers and a small, meaningless function
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

public class Node {

  public short left = Short.MAX_VALUE;
  public short right = Short.MAX_VALUE;

  public boolean areIdentical(){
    if (left == right){
      right = Short.MAX_VALUE;
      return true;
    } else {
      return false;
    }
  }
}
```

C.2.18 DummyTypeInterface

```
package openKernel.objectManager;
/**
 * A fake TypeInterface to show the kind of type data I need.
 *
 * A real TypeInterface would probably have to query the compile-time
 * system for this information. It's possible that if this querying
 * were slow, then type-data could be cached.
 *
 * @version  0.2, June 1997
 * @author   stuart yeates
 */

class DummyTypeInterface extends TypeInterface {

  /**
   * getOutGoingReferences returns an array of indexes into the
   * Workspace indicating the targets of outwards references in
   * object
   */
  final short[] getOutGoingPointers(MemoryObject object){
    return object.getReferences();
  }
}
```

C.3 Other Classes

The following classes are not my own work, but are parts of the **OpenKernel** system.

C.3.1 Iterator

```
// Iterator.java

/**
 * The <tt>Iterator</tt> interface is an abstract class that defines a
 * sequential traversal for a container object without exposing its
 * implementation.
 *
 * @version  0.1,  Dec 20, 1996
 * @author   Michel de Champlain
 * @see      Gamma et al. - Iterator pattern (p. 257)                    10
 */

package openKernel.objectManager;

public interface Iterator {

  /** @section Modifiers */

  /**
   * Positions the iterator to the first object in the container.        20
   */
  public void     reset();

  /**
   * Positions the iterator to the next object in the sequence.
   */
  public void     next();

  /** @section Selectors */
                                                                         30
  /**
   * Returns the index at the current position in the sequence.
   */
  public short    current();

  /**
   * Checks whether the index refers to an object within the container.
   * @return   <tt>true</tt> when there are no more objects in the sequence.
   */
  public boolean  isDone();                                              40
}
```

C.3.2 MemeoryObject

```
// MemoryObject.java

/**
 * The <tt>MemoryObject</tt> class is a proxy that
 * provides support for the garbage collector and a memory object link
 * used by several containers.
 *
```

```
 * @version  0.2, June 1997
 * @author   stuart yeates
 * @author   Michel de Champlain
 */

package openKernel.objectManager;

public class MemoryObject {
  MemoryObject() {
    _next = _prev = _id   = Short.MAX_VALUE;
    _tag  = _type = _prio = Byte.MAX_VALUE;
    _o = null;
  }

  /**
   * getReferences - return an array of the references in this object
   */
  short[] getReferences(){ return NULLARRAY; };
  private final static short[] NULLARRAY = { };

  short  _next;  // two fields for Circlist
  short  _prev;  //
  short  _id;    // this objects place in the Workspace._memory
  byte   _tag;   // mark bits for the garbage collector
  byte   _type;  // the type of the _o
  byte   _prio;  // unused
  Object _o;     // a pointer to the object for which this is proxy
}
```

C.3.3 Workspace

```
// Workspace.java

/**
 * The <tt>Workspace</tt> class is a singleton pattern that ensures only
 * one instance of the workspace in memory and provides a global point
 * access to it.
 *
 * This class also makes instances of all memory objects (or proxies).
 *
 * Limitations:
 *   MAX_OBJECTS = maximum of memory objects that can be allocated.
 *
 * @version  0.1,   1 Dec 1996
 * @author   Michel de Champlain
 * @see      Gamma et al. - Singleton pattern (p. 127)
 */

package openKernel.objectManager;

public class Workspace {
//  public static final short MAX_OBJECTS = 32767;
  public static final short MAX_OBJECTS = 1000;
```

```
  public static Workspace instance() {
    if (_instance == null) _instance = new Workspace();
    return _instance;
  }

  public short nMemoryObjects() {
    return (short)_memory.length;                                    30
  }

  public MemoryObject[] memory() {
    return _memory;
  }

  public MemoryObject get() {
    return descs.count() != 0 ? _memory[ descs.get() ] : null;
  }
                                                                     40
  public void put(MemoryObject mo) {
    descs.put( mo._id );
  }

  private Workspace() {
    _memory = new MemoryObject[ MAX_OBJECTS ];

    descs   = new DescriptorStack( MAX_OBJECTS );
    i       = descs.newIterator();
                                                                     50
    for (short n = 0; n < _memory.length; n++) {
      _memory[ n ] = new MemoryObject();
      _memory[ n ]._id = n;
    }
  }

  private static Workspace       _instance = null;
//  private static MemoryObject[] _memory   = null;
          static MemoryObject[] _memory   = null;
                                                                     60
  private static DescriptorStack   descs;
  private static Iterator          i;
}
```

C.3.4 AbstractCirclist

```
// AbstractCirclist.java

/**
 * The <tt>Stackable</tt> interface is an abstract class that defines a
 * standard stack interface for different stack implementations.
 * To enable polymorphic iteration, <tt>Stackable</tt> defines a
 * factory method <tt>newIterator()</tt>, which subclasses override
 * to return their corresponding iterator.
 *
 * @version  0.1,  Dec 20, 1996                                      10
 * @author   Michel de Champlain
```

```
 * @see       Gamma et al. - Iterator pattern (p. 257)
 * @see       Gamma et al. - Factory method (p. 107)
 */

package openKernel.objectManager;

public interface AbstractCirclist {

  /** @section Constructor */

  /**
   * Creates an iterator to support polymorphic iteration.
   */
  Iterator  newIterator();

  /** @section Modifiers */

  /**
   * Pushes the index onto the stack.
   */
  void      putFirst(MemoryObject index);
  void      putLast(MemoryObject index);

  /**
   * Pops the top index from the stack.
   */
  MemoryObject getFirst();

  /** @section Selectors */

  /**
   * Returns the number of objects in the stack.
   */
  short     count();

  /**
   * Returns the next index followed by thisIndex.
   */
  short     next(short thisIndex);

  /**
   * Returns (without removing) the top index from the stack.
   */
  short     first();
}
```