## Design of a Garbage Collector Using Design Patterns

Stuart A. Yeates\* and Michel de Champlain†
Department of Computer Science
University of Canterbury, New Zealand

This this paper appeared in the proceedings of TOOLS Pacific '97, Melbourne, Australia, 24-27 November 1997

#### **Abstract**

We present six design patterns—Adapter, Facade, Iterator, Proxy, RootSet and TriColour—found during a review of four different garbage collectors. We also capture the design decisions and trade-offs behind the low-level implementation that characterises most garbage collectors. A garbage collector for real-time applications is then designed using the design patterns. We discuss the selected algorithm and various implementation techniques. Finally the performance of the collector is examined using formal methods. This paper presents a novel attempt to "mine" and capture the essential design decisions and trade-offs in garbage collectors.

#### 1 Introduction

Originally created for Lisp, garbage collection is commonly used in object oriented languages<sup>1</sup> and functional languages<sup>2</sup>, where it simplifies memory management and increases encapsulation by hiding object lifecycles.

Since the 1970's several algorithms have been developed, many of which are 'tracing' or 'sweeping' algorithms; that is they trace references between objects in the heap to find which are no longer reachable by the application. The research described here has deliberately been kept general to ensure that the resulting design patterns can be used for this entire family of garbage collection algorithms, including Mark-and-Sweep, Mark-and-Compact, Semi-Space Copying and Generational variants.

Traditionally, due to the over-riding need for speed, garbage collectors for objectoriented systems have been written in assembler or highly-optimised C. While writing garbage collectors in such ways produces fast collectors, it inhibits flexibility, preventing experimentation and making tuning difficult. When this research started (early 1996) there were no publicly available garbage collectors written in object-oriented languages. Modern surveys of the garbage collection field can be found in (Corporaal and Veldman, 1991; Wilson, 1992; Nilsen, 1994), and (Wilson et al., 1995) present an extensive bibliography. (Jones and Lins, 1996) presents a thorough exploration of the field including implementation techniques and code samples.

The main contribution of this paper is an integration of two different areas: garbage collector and design patterns. It is also a novel attempt in "pattern mining" to capture the essential design decisions and trade-offs in garbage collectors. In Section 2 we discuss briefly the garbage collectors and the design patterns found in them, with focus on the two

<sup>\*</sup>Current address: Trimble Navigation, Christchurch, New Zealand

<sup>&</sup>lt;sup>†</sup>Address after December '97: Department of Electrical and Computer Engineering, Concordia University, 1455 de Maisonneuve Blvd. West, Montréal, Québec, Canada H3G 1M8. michel@ece.concordia.ca

<sup>&</sup>lt;sup>1</sup> Such as Java, Oberon, Sather and Smalltalk

<sup>&</sup>lt;sup>2</sup>Such as HUGS, SML and Scheme

new patterns TriColour and RootSet. In Section 3 we give the design of a garbage collector built using these patterns. In Section 4 we present a sample of the algorithmic analysis of the design, explaining the important features. Lastly, Section 5 summarises the conculsions of this paper.

### 2 Design Patterns and Garbage Collection

Four garbage collectors, each with a different target language, were examined for design patterns. The Tolpin collector is a nonincremental 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 (Boehm and Weiser, 1988) is a general purpose collector for C/C++ with no access to type information. The Baker78 collector (Baker, 1978) 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), an abortable, nonincremental, mark-and-sweep collector with access to complete type information.

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) domain specific patterns—those unique to 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 Table 1 summarises which of the patterns were found in the collectors reviewed.

# 2.1 General Patterns in Garbage Collection

The general patterns found were the adapter, facade, iterator and proxy patterns, each of which were put to specific uses in garbage collection. These patterns will not be discussed in detail<sup>3</sup>. The specific patterns were the RootSet and tricolour patterns and their description here represents original work.

#### **Adapter and Facade Patterns**

Adapters and facades are common in garbage collection, and in some situations the distinction between them is not clear. Functionally, adapters and facades each provide types of flexibility at the same point, the interface between the application and the garbage collector. Adapters allow subsystems to change their interface syntax independently, while facades allow subsystems to change their internal decomposition independently.

#### **Iterator Patterns**

Iterators are objects which allow iteration over an aggregate object without exposing its internal structure. The primary action of all nonreference 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.

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

- 1. Iterations over roots—the set of pointers into the heap (or a generation) from outside. This iteration usually involves finding all pointers in global data structures and runtime stacks, and can be hard to incrementalise. This 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 roots during the flip() operation and its completion indicates

<sup>&</sup>lt;sup>3</sup> For more details of both groups, see (Yeates and de Champlain, 1997; Yeates, 1997).

Collector	Target	Adapter	Facade	Iterator	Proxy	TriColour	RootSet
	language						
Baker78	LISP	no	no	yes	yes	yes	no
Boehm	C/C++	yes	yes	yes	yes	yes	yes
Tolpin	Oberon-2	no	no	yes	yes	no	yes
Java	Java	no	yes	yes	yes	no	no

Table 1: The patterns found in each of the collectors

the end of the garbage collection cycle. This occurs once per garbage collection cycle.

 Iterations over pointers within an object (scanning)—these are used to find which other objects a particular object references.

#### **Proxy Patterns**

Proxies are used in garbage collectors in three ways:

- 1. To control access to the objects they guard. They are used to implement readand 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 objects they guard. They can, for example, 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.

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. Secondly by storing the proxy with the object (which lends itself more readily to incrementalisation of proxy initialisation and re-sizing 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.

#### 2.2 TriColour Pattern

TriColour marking is the theoretical proofof-correctness on which all known incremental sweeping garbage collection rests (Dijkstra *et al.*, 1978). The proof involves moving objects between three sets, or 'colours,' hence the name. 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.

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 TriColour marking or an equivalent data structure. Non-tracing (pure reference counting) collectors incorporate neither TriColour nor an equivalent data structure.

Name TriColour.

**Intent** Maintain the state upon which the Tri-Colour proof-of-correctness rests.

Motivation The TriColour 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 TriColour marking proof as an abstract data type, decou-

pling the proof-of-correctness from the **Solution** Create implementation of the collection.

**Applicability** All known tracing incremental garbage collectors use the TriColour marking approach.

Consequences In both the incremental garbage collectors examined, the TriColour featured far more prominently in the system description than the implementation. By explicitly embedding the theoretical proof in 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.

#### 2.3 RootSet Pattern

Roots, pointers into the heap or a generation, are the starting points for the collectors' 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 with 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 is within the garbage collector (for example the Tolpin collector). Dealing with multiple sources of roots unacceptable complicates the TriColour.

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

Consequences 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, as well as other sources of roots.

### 3 Collector Design

A garbage collector was designed and implemented for **OpenKernel** a real-time object-based micro-kernel (de Champlain, 1996) using design patterns and written in Java. The garbage collector was specified and designed using object-oriented techniques and design patterns. The garbage collector attempts to perform in real-time, and is de-coupled from the memory allocation and type subsystems.

#### 3.1 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.
   Due to the high-level programming language (Java) used in this project exact times were not calculated, but the ability to translate the algorithms directly into a low level language and guarantee response times measured in the millisecond range was important.
- Avoiding library calls of dubious response 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<sup>4</sup>—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.

#### 3.2 Design

Figures 1 and 2 shows the initial class decomposition diagram for the garbage collector with the following classes:

- Collector—the interface to the garbage collection sub-system. All interactions between the garbage collector and other sub-systems (other than the typeinterface) 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 Tri-Colour, the primary location of garbage collection state. There are as many

<sup>&</sup>lt;sup>4</sup>Finalisers are closely related to C++ destructors.

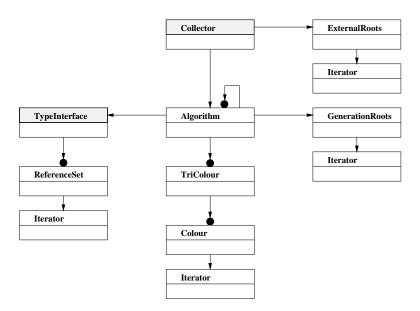


Figure 1: The initial class decomposition diagram of the objects in the garbage collector.

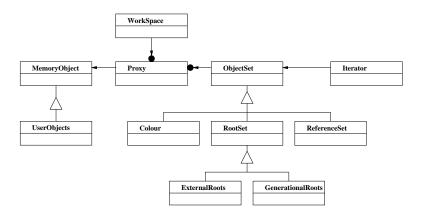


Figure 2: Class decomposition diagram showing ObjectSet and related classes. Note that due to the simple generational model adopted, GenerationalRoots are not used in the implementation.

TriColours in a garbage collector as there are generations (one for nongenerational 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 on TriColours are register(), deRegister(), isMember(), isGrey(), isWhite(), markGrey(), isBlack(), mark-White(), 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 2). The aggregates are
  - Colour—an aggregate of all the objects of a single colour in a TriColour. Valid operations on Colours are isNotEmpty(), isEmpty(), newIterator(), remove(object), insert(object) and next().
  - ReferenceSet—an aggregate of references within a heap object. A Java native array with the extra valid operation newIterator().
  - RootSet—an aggregate of objects to which are 'roots', one of:
    - \* 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 our implementation.

- Iterator—an iterator over an ObjectSet.
- MemoryObject—the class representing all user objects. All access to Memory-Objects is via a Proxy, a pointer to the stored in the Workspace.
- Workspace—a singleton object responsible for allocating MemoryObjects, contains an array of references to all application-visible MemoryObject. This relationship is elaborated on in the example given in subsection 2.1.

However, after an examination of the properties of the various data structures (see subsection 3.3.2, it became increasingly clear that the overhead of inter-generational roots was likely to become high. For this reason, a very simple generational model was used, in which all objects which survive a flip() are promoted. When working in an 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. The proof, however, only calls for an 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 its Iterator into one object saves two objects per TriColour, and several object interactions. This limitation of one Iterator per Colour is similar to Eiffel's cursor (Meyer, 1994).

Figure 3 shows Figure 1 updated for these implementation decisions.

# 3.3 Implementation and Performance Issues

#### 3.3.1 Choosing Algorithms

Two major criteria affected our 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 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 cannot be incrementalised. Non-Treadmill Semi-Space Copying algorithms (Wilson, 1992) would both require a read barrier and handle pinned memory memory accessed by system components unable to use proxies, for example direct memory access hardware—only with considerable added complexity. 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 its 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<sup>5</sup>, this guarantee can be provided. If implemented in its 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 we concluded that the most suitable algorithm to implement was a generational mark and sweep garbage collector.

#### 3.3.2 Choosing 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 with 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 soft 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 collector (if any).

<sup>&</sup>lt;sup>5</sup>Generally tracing two objects per object allocation, but always calculable and calculated at, or before, compile time

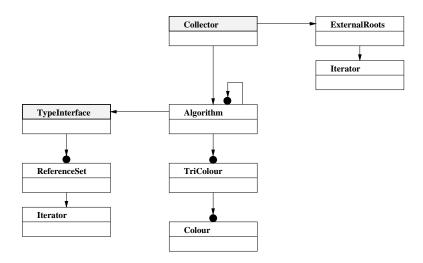


Figure 3: The final class decomposition diagram of the objects in the garbage collector, after the generational mechanism has been removed, and Colour and its Iterator have been united.

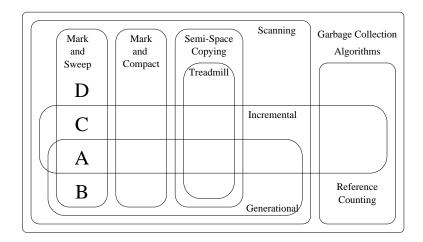


Figure 4: A Venn diagram showing the relationships between the main garbage collection algorithms

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. Several of the structures had internal (int) and external (ext) variants, the difference being shown in Figure 5.

Each was 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 2.

The number of such data structures within a generation collector is also important. In this respect, we approximate that we need one RootSet of pointers into the heap from elsewhere, internal data structures, which don't allow multiple membership, since there may be an unbounded number of roots pointing to a single object.

We also need at least 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, External-Roots 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.

#### 3.3.3 Using 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.

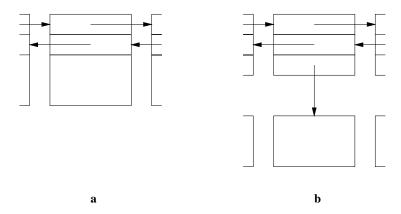


Figure 5: Interior (a) and exterior (b) doubly linked lists. In a, pointers for the list are in the same data structure as the data, while in b the pointers are in a separate data structure with a pointer to the data structure containing the data.

Data Structure	membership	multiple	insert	remove	join	overhead
	test	membership				(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)	<b>O</b> ( <i>m</i> )	Cm

Table 2: 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, p is the size of a pointer to an object or an index to a proxy (1–4 bytes) and p0 is the size of the counter used. All objects are assumed to have an overhead of p0, 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.

#### 3.3.4 Finding Pointers

It 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.

#### 3.3.5 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 better when they have many objects in them, increasing the probability that two free 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. We 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.

## 4 Algorithmic Analysis

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.current-TimeMillis(). During testing, however, we experienced problems with the timing of results,

and timing was abandoned.

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 (Aho and Ullman, 1995), as is necessary in real-time systems, because in systems of non-trivial complexity it is extremely hard to exercise all execution paths through the source, and even then ensuring the worstcase case been seen can be difficult.

A detailed algorithmic analysis<sup>6</sup> has been performed on 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—the only loop was in GenAlgorithm.trace() (see Figure 3).
- the scarcity of library calls—the only necessary library call was to java.lang.reflect.Array.getLength(), to determine the length of arrays. Use of this library call was optional, the alternative being a user-written array package with the array length at the start of the array.

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

<sup>&</sup>lt;sup>6</sup> For a more details of this analysis see (Yeates, 1997).

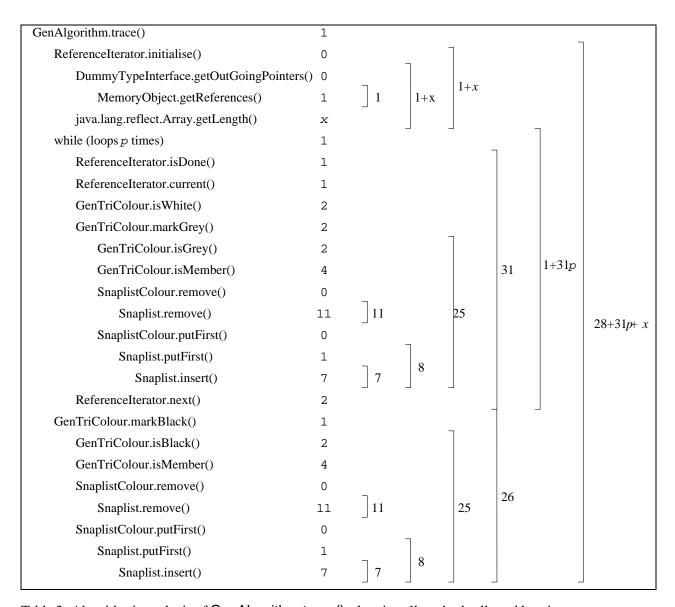


Table 3: Algorithmic analysis of GenAlgorithm.trace(), showing all method calls and looping structures. The method call java.lang.reflect.Array.getLength(), which is part of the Java runtime system, is a somewhat known factor. However, its execution time (x) is likely to be very low as a call to this 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.

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).

The time taken for all collector operations may is calculated in terms of p and x. Method call overhead is assumed to be negligible, since all methods (other than java.lang.reflect.Array.getLength(), which has already been discussed) are within the same module, and all methods are very short. It is anticipated that in preparation for production use, all garbage collection code could be moved into a single java compilation unit and all the classes made private inner-classes of the garbage collector interface. This optimisation, which could be automated, would have the same effect as declaring all classes and methods final.

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).

The cost of registering a single object (i.e. during a new()) 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 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 it appears that the LinkedListRootSet implementation of RootSet is unlikely to be used in a real system, as 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, "piggybacking" on the normal stack frame pointers, rather than maintaining a separate data structure.

Table 4 shows the worst case performance of each of the collector algorithms and Root-Set implementations. In each case, add-Root() and removeRoot() are inserts and removes in the respective data structures (see table 2).

The operations are given in table 4 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 performed in O(1) time.

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

Table 4: 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.

#### 5 Conclusion

In this paper, we have summarised and integrated the most central aspects of two different areas: garbage collector and design patterns. We have examined four garbage collectors and captured six design patterns Adapter, Facade, Iterator, Proxy, RootSet and TriColour. The last two of which, RootSet and TriColour, are new patterns, domain specific to garbage collection, whose description here represents original work. All of these patterns were defined and discussed in a manner independent of the garbage collection algorithm in use. A requirements analysis was performed for the proposed collector, running in realtime was drawn up, and a collector designed using these requirements and the design patterns. An appropriate algorithm was selected and implemented. A range of implementation techniques were discussed. A performance analysis was performed, showing that performance was acceptable in all but one method. A scheme to rectify this, at the cost of considerably closer coupling between the execution stack and the collector, is proposed.

We believe that our analysis of existing designs will help advance research efforts while also providing guidance for designers. In our future research, we plan to apply these patterns—even if they provide less efficient implementations—as a more flexible way of reusing and experimenting garbage collectors in programming languages.

## Acknowledgements

We would like to thank the anonymous reviewers for their helpful suggestions and comments.

#### References

- Alfred V. Aho and Jeffrey D Ullman. *Foundations of Computer Science*. Freeman, C edition, 1995.
- Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.
- Henry G. Baker. The treadmill: Real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3):66–69, March 1992.
- Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, Sept 1988.
- F. Buschmann, R. Meunier, H. Robnert,
   P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture: A system of patterns. Wiley, 1996.
- H. Corporaal and T. Veldman. The design space of garbage collection. In *Proceedings, Advanced Computer Technology, Reliable Systems and Applications, 5th annual European computer conference.*, pages 423–428, 1991.
- M. de Champlain. Patterns to Ease the Port of Micro-kernels in Embedded Systems. In *Proc. of the 3rd Annual Conference on Pattern Languages of Programs (PLoP'96), Allerton Park, IL*, June 1996. Published as a tecnhical report of Washington University.
- Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An

- exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP'93 Object-Oriented Programming*, Kaiserslautern, Germany, 1993.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Richard Jones and Rafael Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, 1996.
- Thomas Kofler. Robust iterators in ET++. *Structured Programming*, 14:62–85, March 1993.
- Bertrand Meyer. Reusable Software: The Base object-oriented component librarires. Prentice Hall, 1994.
- Kelvin D. Nilsen. Reliable real-time garbage collection of C<sup>++</sup>. *Computing Systems*, 7(4):467–503, fall 1994.
- Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry G. Baker, editor, *Proceedings of the 1995 International Workshop of Memory Management*, number 986 in Lecture Notes in Computer Science, pages 1–117. Springer, September 1995.
- Paul R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Proceedings of the 1992 International Workshop of Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42. Springer, September 1992.
- Stuart A. Yeates and Michel de Champlain. Design patterns in garbage collection. In Robert S. Hanmer and Don Roberts, editors, *Proc. of the 4rd Annual Conference on Pattern Languages of Programs*

- (*PLoP'97*), Monticello, Illinois, USA, 1997. Published as technical report # wucs-97-34 of Washington University.
- Stuart Yeates. Design patterns in garbage collection. M.Sc (Comp. Sci.), University of Canterbury, Christchurch, New Zealand, June 1997.