Design Patterns in Garbage Collection

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

October 16, 1996

An earlier version of this paper appeared in the proceedings of the fourth annual Pattern Languages of Programming (PLoP) conference, held at Allerton Park Illinois, 2-5 September 1997

Abstract

This paper describes several design patterns found in garbage collectors. The patterns we present are divided into two groups. The first group are two new design patterns: *Rootset* and *TriColour* that have been used in the garbage collection domain for up to 20 years. The second group of patterns are reported in the GoF book, such as *Adapter*, *Facade*, *Iterator* and *Proxy* ,but we examine their use in the garbage collection domain. These patterns can be used by language implementors to provide a less efficient, but simpler and more flexible way of implementing and reusing garbage collectors in programming languages than current low-level and nonportable methods.

1 Introduction and Background

Garbage collection is the automatic recovery of resources, usually main memory, using language-level constructs. Garbage collectors have traditionally been written in hand coded assembler, making them hard to optimise, difficult to port or reuse (between target operating systems or client languages) and almost impossible to experiment with. The problem we were attempt to solve in this paper is that of how we can experiment and reuse garbage collectors—and their different algorithms.

Since the 1970's several algorithms have been developed, many of which 'tracing' algorithms; that is they trace pointers 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.

*Current address: Trimble Navigation, Christchurch, New Zealand. `stuart.yeates@trimble.co.nz`
†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`

The Mark-and-Sweep algorithm is the original tracing algorithm. Incremental algorithms split the collection across many small increments to avoid a pause long enough for users to notice. Compacting algorithms move objects in memory to defragment it. Semi-Space and Generational collectors partition objects by age enabling them to focus on young objects, as most objects die young (Wilson, 1992).

This paper presents design patterns found in four garbage collectors, each with different target languages but similar 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 (Boehm & 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 which 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), non-incremental mark-and-sweep collector with access to complete type information.

These collectors were examined as a step in the acquisition of domain knowledge for the construction of a garbage collector by the authors. Design patterns (Gamma *et al.*, 1995) were used as a tool to capture this relevant domain knowledge. Because the authors wished to postpone the selection of a garbage collection algorithm, the decision was made to separate the design patterns from the algorithms they supported, for this reason, the design patterns are presented here make no mention of the algorithms and all but TriColour (which is only using in incremental algorithms), are applicable to all sweeping (non-reference counting) garbage collectors.

The design patterns captured in the garbage collectors examined fall into two groups: (1) specific patterns—those unique to a particular domain, in this case, garbage collection, and (2) general patterns—those commonly found in both software and the literature on design patterns, such as those documented in (Gamma *et al.*, 1995). 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 specific patterns were the rootset and tricolour patterns are introduced here for the first time. The general patterns found were the adapter, facade, iterator and proxy patterns, each of which were put to specific uses in garbage collection.

One pattern we didn't find that we had thought we might was the strategy pattern in which a family of algorithms is encapsulated so that the algorithms may be varied independently of their clients. None of the collectors examined displayed significant separation of algorithms from the rest of the collector, a necessary requirement for the strategy pattern.

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

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

1.1 TriColour

The TriColour marking is the theoretical proof of correctness on which all known incremental sweeping garbage collection rests (Dijkstra *et al.*, 1978). In the proof, objects are moved between three sets, black (reachable, live, objects), grey (reachable objects which may contain references to objects of unknown reachability) and white (objects of unknown reachability), 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 (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 TriColour marking or an equivalent data structure. Non-sweeping (pure reference counting) collectors incorporate neither TriColour nor an equivalent data structure.

Name TriColour.

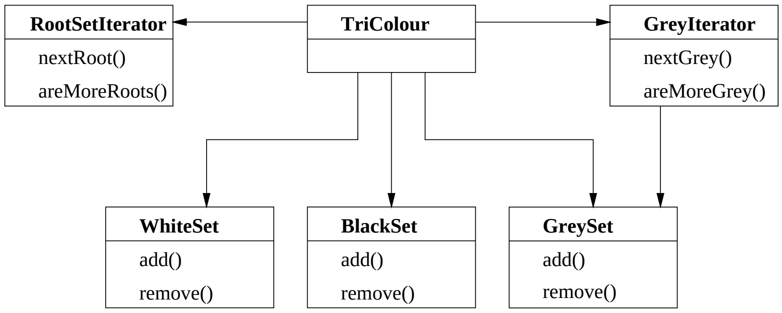
Intent Maintain the state upon which the tri-colour proof-of-correctness rests.

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. To be maintainable, important features of the code need to be obvious and documented, or risk being broken.

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

Applicability All known incremental garbage collectors use the tri-colour proof-of-correctness in some form, and hence the TriColour pattern.

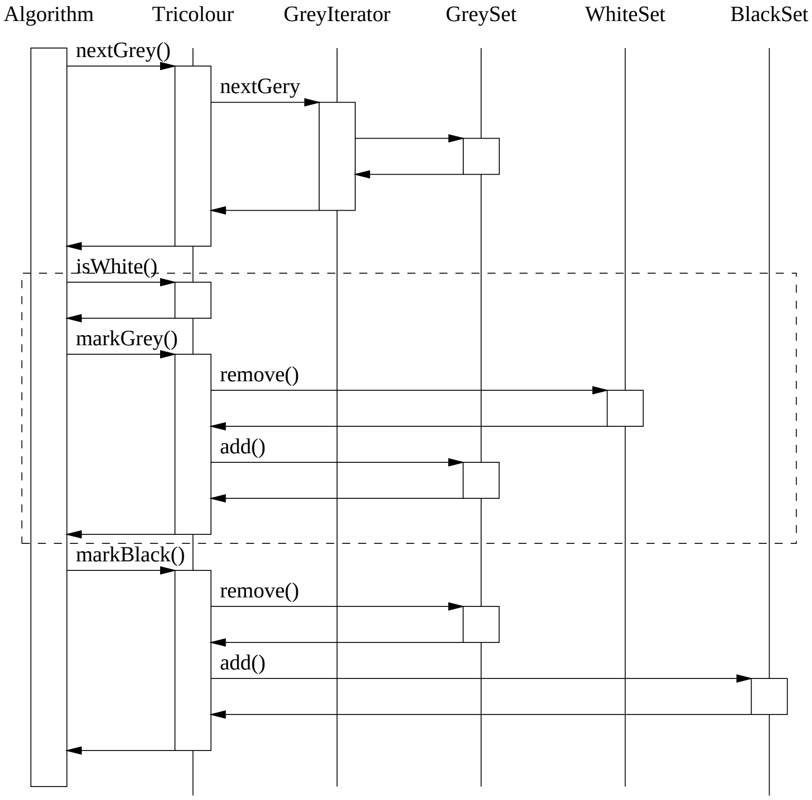
Structure



Participants

- TriColour
  - responsible for maintenance of state for the tri-colour proof-of-correctness.
- RootSetIterator
  - the source of roots of the iteration over the heap, an Iterator over a RootSet.
- 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 determines which object to trace with a call of nextGrey, which is forwarded to GreyIterator, which in turn forwards it to 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 (these references are obtained using an iterator which is not shown). 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 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. The TriColour pattern is likely to add a level of indirection at several places, including both the allocation and tracing routines (these are the 'inner loop' of the garbage collection). In both cases this level of indirection, however, is a method of a single operation (another method call), so should be easy to optimise in production code.

Implementation Past implementations have not implemented the tricolour directly, describing it instead in their documentation, and presented code which contained the tricolour only implicitly in a convoluted form. 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 differs from the basic tri-colour as outlined by (Baker, 1978) by also including management of unreachable, but unreclaimed objects. This allows 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 fourth set of objects (which are genuinely free and available for reuse) in that by the end of the collection all have been returned to the external memory manager. 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.
 *
 * 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 ? */
  boolean isMember(Object object);

  /**is this object marked grey ? */
  boolean isGrey(Object object);

  /** is this object marked white ? */
  boolean isWhite(Object object);

  /** is this object marked black ? */
  boolean isBlack(Object object);

  /** mark an object grey */
  void markGrey(Object object);

  /** mark an object white */
  void markWhite(Object object);

  /** mark an object black */
  void markBlack(Object object);

  /** are there more grey objects ? */
  boolean areMoreGrey();

  /** which is the next grey item ? */
  Object nextGrey();

  /** start a new garbage collection cycle  */
  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.
   */
  void deRegister(Object object);
}
```

1.2 RootSet

Roots, pointers into the heap or a generation, 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.

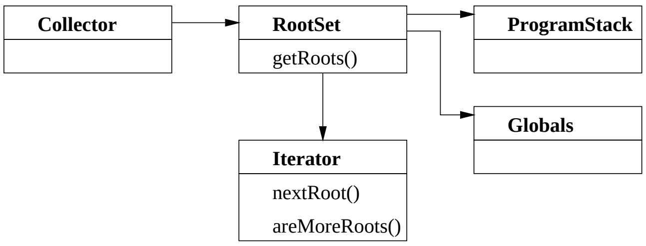
Intent Abstract the generation and retargeting of RootSets.

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). 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.

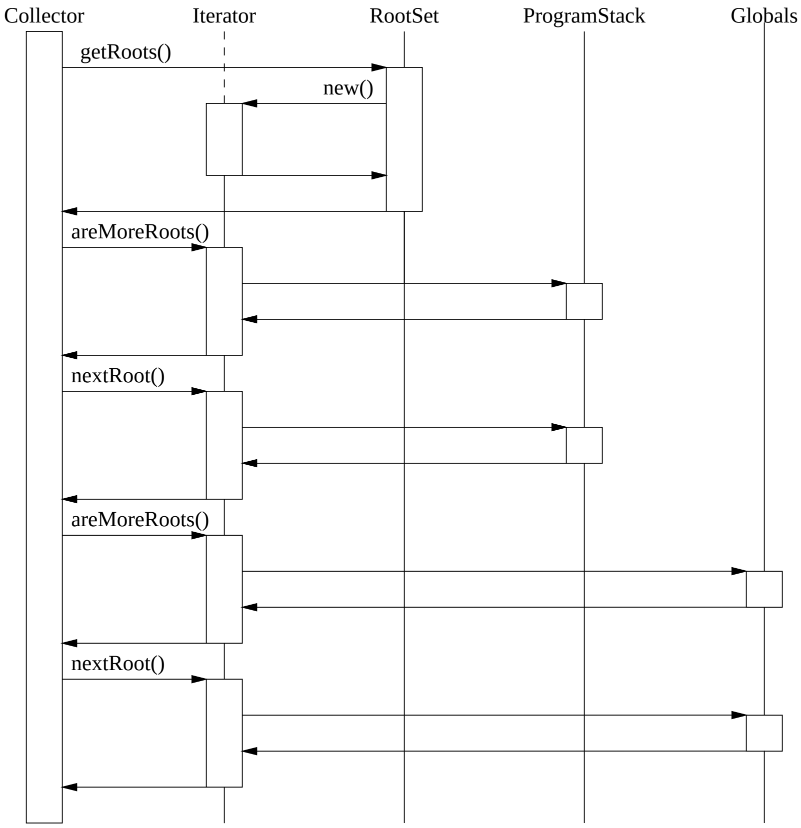
Structure



Participants

- Collector
  - needs to iterate over all the roots in the heap or a generation 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, as well as other sources of roots. As with the TriColour, the RootSet is likely introduce a level of direction, again the methods a very simple and suitable targets for optimisation. The RootSet indirection is not the the garbage collector's 'inner loop,' but in the flip(), the largest single operation which cannot be incrementalised. For this reason, in hard real-time collectors, where the maximum latency is the critical factor, the RootSet may need to be optimised in production code.

Implementation RootSet is effectively just a standard interface to a varying collection of data structures and a single iterator over them. 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 possible granularities for interface between the program execution stack and the garbage collector:

1. per frame—whenever a stack frame is pushed or popped, the heap from that frame are added to the RootSet. The cost of this method is extremely high, 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 structures. The Toplin 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 entirely incurred during the flip() operation, increasing the maximum latency. The Boehm collector uses this method. 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.
3. per program—a single root is used. This is the trivial RootSet of those implementations which allocate the stack frames 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 our implementation of a garbage collector in Java.

```java
package openKernel.objectManager;
/**
 * Interface RootSet, the interface between the RootSet and the garbage
 * collector.
 *
 * 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();

}
```

1.3 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 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.

The key difference between a facade and an adapter is that a facade provides a single integrated interface 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 originally designed for. Given the evolving nature of the Baker78 and Boehm collectors, and the large numbers of languages with similar but different memory management interfaces (C, C++, Pascal, Modula-2, etc), the distinction is necessarily not always clear.

1.3.1 Adapter

Adapters are common in memory management, for example, the Boehm collector, originally for C, uses an adapter to enable the same collector 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).

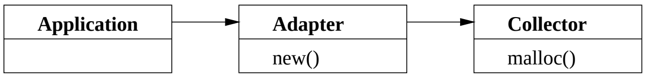
Intent Decouple the interface between the garbage collector and external system components.

Motivation 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.

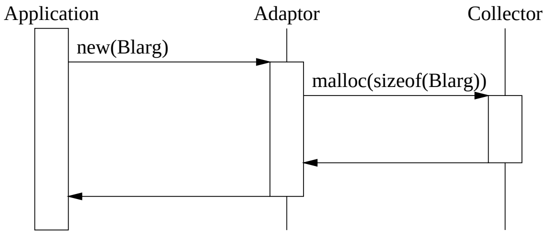
Structure



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 the appropriate C function, each method has explicit parameter and return types, so type errors are caught at the interface to the collector at compile-time.

```
class GC {
```

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

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

1.3.2 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 encapsulation at the package level 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 client application for encapsulation, namespace conservation and to accommodate the dropping of garbage collectors into systems designed to utilise traditional memory management systems.

Name Facade.

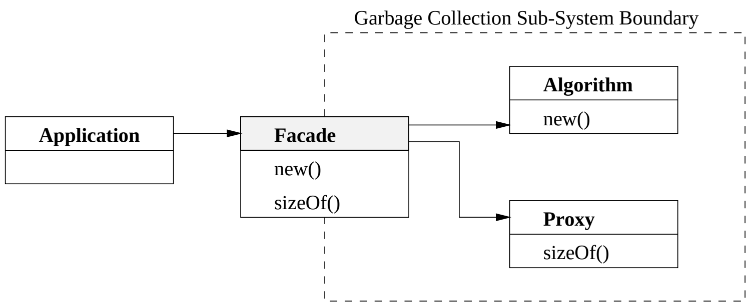
Intent Decouple interface between the garbage collector and external system components.

Motivation 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 on design freedom.

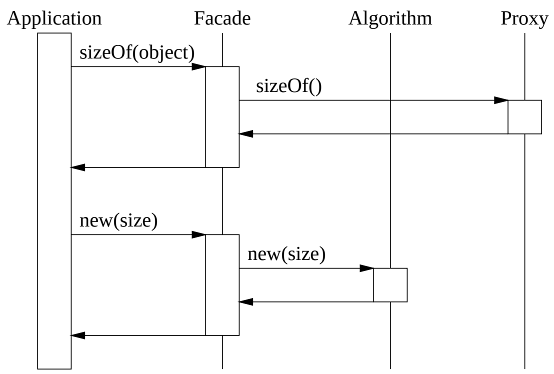
Structure



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, as with the adaptor, 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;

/* 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;

/* 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);

/* 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 */
```

1.4 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, the incremental collectors recursive state being held in the TriColour rather than on the execution stack.

The traversal order (depth-first, breadth-first, hill-climbing or the more domain-specific hierarchical traversal) has important 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 may 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 (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 the end of the garbage collection cycle. The tricolour holds the state of this iteration, and the read-barrier requires robustness from the iterator. This occurs once per garbage collection cycle, but is commonly performed in very small increments.

3. Iterations over pointers within an object (scanning)—these are used to find which other objects a particular object references. This 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 predictable, they can be reinitialised and reused. This eliminates all need to create new iterators during a collection cycle.

Name Iterator.

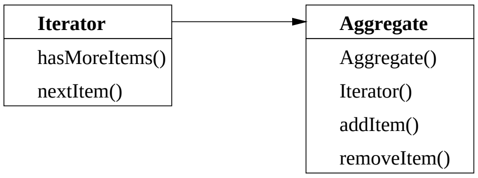
Intent Abstract iteration, iteration over roots, heap objects or references with an object.

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, of Iterators.
- Iterator
  - Maintains a reference to Aggregate
  - Due to the large 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.

Implementation Because of the high number of iterations performed during the course of a garbage collection, the desirability of the garbage collector not using temporary objects (since the garbage collector is likely to be invoked when there is little or no memory for their creation), reuse of Iterator objects 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();
  }
}
```

1.5 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). Such an implementation requires a level of indirection on every object access. 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. (Wilson *et al.*, 1995)

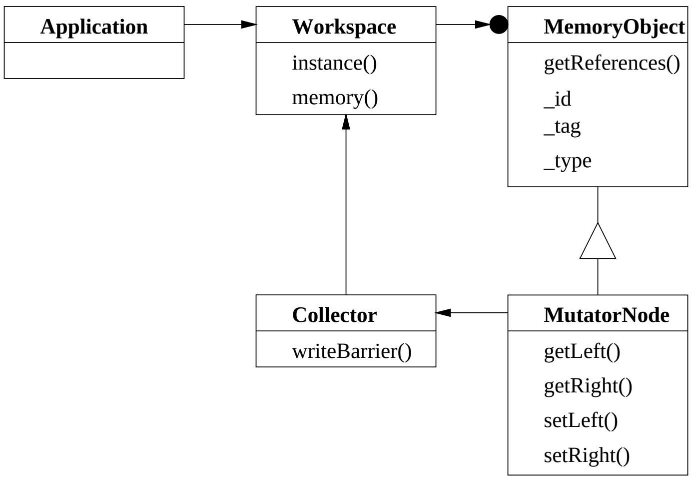
Name Proxy

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 our 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 Potentially, 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 our implementation of a garbage collector in Java. Workspace is a singleton 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)

Each of the entries in the array is an atrophied proxy: it is a pointer to the real object, which is held by objects in place of a pointer to the real object.

```
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;
  }
  private Workspace() {
    _memory = new MemoryObject[ MAX_OBJECTS ];
  }
  public MemoryObject[] memory() {
    return _memory;
  }
}
```

MemoryObject is the base class of all heap objects, it has a methods for retrieval of references to other heap objects within the object, the location of the object within Workspace._memory, the type of the object and several mark bits. Java implements arrays as an array of references to objects, 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 nullify, set or change the elements in Workspace._memory, during object deallocation, 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;
    /* .... */
    memory[s] = anotherObject;
    /* .... */
  }
  /* .... */
}
```

---

2 Conclusion

We have examined four garbage collectors and found six patterns shared by two or more of them. This is evidence of an area of design commonality suitable for capturing in design patterns. We are currently implementing a garbage collector in Java using these patterns and hope to confirm their usefulness in aiding this task.

3 Acknowledgements

We would like to thank Kyle Brown, Ken Auer, Manish Bhatt, John Brant, Jim Doble, Neil Harrison, Philippe Lalanda, Eugene Wallingford and the others who gave valuable feedback at PLoP'97.

References

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

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

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

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

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

Buschmann, F., Meunier, R., Robnert, H., Sommerlad, P., & Stal, M. 1996. *Pattern-Oriented Software Architecture: A system of patterns*. Wiley.

Dijkstra, E. W., Lamport, L., 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.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. *In: ECOOP'93 Object-Oriented Programming*.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

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, J., Joy, B., & Steele, G. 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.

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

Wilson, P., Lam, M., & Moher, T. 1991 (June). Effective "static-graph" Reorganisation to Improve Locality in Garbage-Collected Systems. *Pages 177–191 of: Proceedings of ACM SIGPLAN 1991.*

Wilson, P., Johnstone, M., Neely, M., & Boles, D. 1995. Dynamic Storage Allocation: A Survey and Critical Review. *In:* (Baker, 1995b).

Wilson, Paul R. 1992. Uniprocessor Garbage Collection Techniques. *Pages 1–42 of:* Bekkers, Y., & Cohen, J. (eds), *Proceedings of the 1992 International Workshop of Memory Management.* Lecture Notes in Computer Science, no. 637. Springer.