Are Your Incoming Aliases Really Necessary? Counting the Cost of Object Ownership

Alex Potanin, Monique Damitio, James Noble
Victoria University of Wellington, New Zealand
{alex|monique|kjx}@ecs.vuw.ac.nz

***Abstract*—Object ownership enforces encapsulation within object-oriented programs by forbidding incoming aliases into objects' representations. Many common data structures, such as collections with iterators, require incoming aliases, so there has been much work on relaxing ownership's encapsulation to permit multiple incoming aliases. This research asks the opposite question: Are your aliases really necessary?**

In this paper, we count the cost of programming with strong object encapsulation. We refactored the JDK 5.0 collection classes so that they did not use incoming aliases, following either the owner-as-dominator or the owner-as-accessor encapsulation discipline. We measured the performance time overhead the refactored collections impose on a set of microbenchmarks and on the DaCapo, SPECjbb and SPECjvm benchmark suites. While the microbenchmarks show that individual operations and iterations can be significantly slower on encapsulated collection (especially for owner-as-dominator), we found less than 3% slowdown for owner-as-accessor across the large scale benchmarks.

As a result, we propose that well-known design patterns such as Iterator commonly used by software engineers around the world need to be adjusted to take ownership into account. As most design patterns are used as a building block in constructing larger pieces of software, a small adjustment to respect ownership will not have any impact on the productivity of programmers but will have a huge impact on the quality of the resulting code with respect to aliasing.

I. Introduction

Encapsulation is a crucial attribute of object-oriented programming and design. Object ownership [1] enforces encapsulation by explicitly identifying the internal representation objects: an object *owns* its representation, and owned objects are protected behind the object's interface. An object should act as a "single entry point" for its representation — considering the heap as a graph, an owner dominates the objects that it owns [2] — in other words, there can be no incoming pointers to an owned object that bypass the object's owner. Figure 1 illustrates this: the nodes making up a linked list are owned by the list — while the element data in the list can exist outside the list. The most restrictive form of object ownership, enforced by the ownership types, is both *strong* and *deep*: strong (or prescriptive) because external references to owned objects cannot be used; and deep, because ownership is transitive — an object that owns the list owns the list's links as well.

Strong, deep, ownership offers a number of benefits for program design, generally because of the single entry point property. For example, owned objects may be deleted as soon as their owner becomes inaccessible, supporting real-time memory management [3]. Security checks carried out on the owning

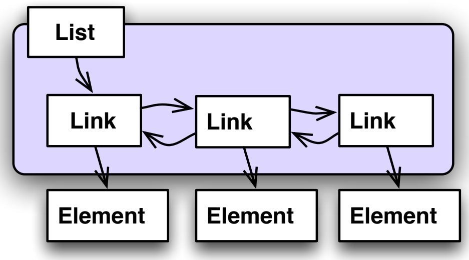

Fig. 1. Encapsulated Linked List

object will always govern access to the owned objects [4]. Class invariants can be affected only by the methods defined in that class and its ancestor classes rather than by method calls on unrelated objects [5].

Ownership is emerging as an important technique for designing parallel and concurrent object-oriented systems to take advantage of multicore processors. Many OO actor systems, including Kilim [6], Thorn [7], and Scala Actors [8], impose an ownership discipline to ensure that actors communicate only via message passing and to prevent one actor accessing another actor's internal representation. The high-throughput pipes-and-filters processing system StreamFlex uses ownership to ensure isolation between filters executed in parallel [9]. Parallel languages such as DPJ [10] or AJ [11] use ownership techniques to describe the data they access and detect interfering computations.

The problem this paper addresses is that adopting an ownership or encapsulation discipline may impose runtime costs on programs. As with other kinds of types, ownership types can be checked statically or dynamically. Static checks, carried out at compile time, do not impose any *direct* costs on program execution, while dynamic checks will impose some overhead directly. Ownership disciplines can also impose *indirect* costs, by precluding designs that bypass objects' interfaces and refer directly to objects' supposedly hidden internal representations. Figure 2 shows an iterator relying on an incoming alias into the nodes of a list — such an iterator can move directly and efficiently from one link to the next, without any reference to the list object within which the links should be encapsulated. Designs that respect encapsulation, and rely on objects' interfaces, may be less efficient than designs that breach encapsulation with incoming aliases.

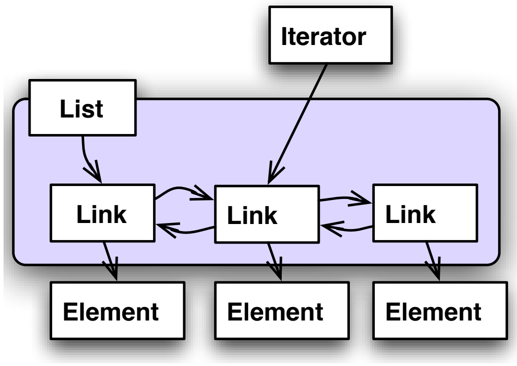

Fig. 2. Linked List with Iterator

In this paper we address the question "*Are your incoming aliases really necessary?*". We study two varieties of strong, deep ownership — owner-as-dominator and owner-as-accessor — and for owner-as-accessor we consider both static and dynamic checking. We then attempt to determine the cost of following each discipline on programs' design and performance. The contributions of this paper are answers to three questions about programming with ownership:

1) How must designs change to respect encapsulation?
2) What performance cost do these changes impose?
3) How does this impact programs' performance?

The remainder of this paper is structured as follows. First, Section II briefly reviews object ownership, and describes the ownership disciplines we investigated for this study. Section III presents a case study of the design of the core collection classes and the refactorings required to adapt them to the object ownership discipline. Section IV describes our benchmarking methodology, and Section V presents the results of both the micro- and macro- benchmarks. Finally Section VI discusses the implications of the results in the context of related work, and Section VII concludes the paper.

II. Ownership

The idea of ownership is to partition the objects accessible from any point in the program according to the object to which they belong [5], [12], [13]. We consider that objects relate to each other in one of three modes: *owned*, *peer*, and *external* objects. An ownership discipline requires programs to keep these modes separate: type casts, assignments, or subsumption must not allow an object in one mode to be accessed as if it were in another mode.

Owned objects are fully private and encapsulated within their owning objects. In Figure 3, object A owns objects B and C, and object C owns D. This is deep ownership because C and D are both owned transitively by A.

Peer objects are siblings with respect to their owning objects, that is, all peers have the same owner. In Figure 3, B and C are peers, and can refer to each other without incoming aliases that would breach encapsulation.

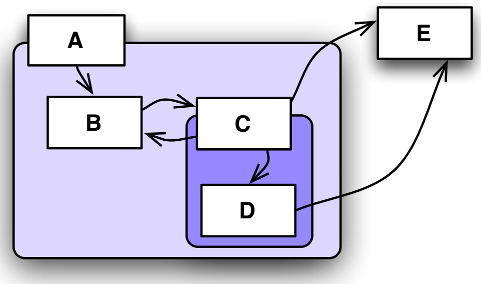

Fig. 3. Relationships between objects with ownership

External objects are — as the name implies — outside the current object, but are still accessible without any incoming aliases. In Figure 3, E is external to all the other objects, and B is external to D. External objects are typically passed in and out as parameters to objects, such as the elements of a collection.

In this study we investigate two realisations of the basic ownership scheme: owner-as-dominator, and owner-as-accessor.

Owner-as-dominator interprets the "no incoming aliases" constraint following Clarke et al.'s original ownership types proposal [1]: neither static nor dynamic references are permitted to cross an encapsulation boundary. In Figure 4, only references shown by the solid black lines are permitted: from external objects to their peers; from an owner to the objects it owns; and between internal peers with the same owner. Aliases that cross encapsulation boundaries (e.g. the dashed red line in Figure 4) are forbidden. Considering the heap as a rooted graph, an owner is a graph-theoretic dominator of all the objects it owns.

Owner-as-accessor takes a different interpretation of the "no incoming aliases" rule, based on Müller et al.'s Universe Types [14], and an experimental dynamically-checked ownership scheme [15]. Owner-as-accessor permits incoming heap references — such as the dashed red line in Figure 4 — provided they are not used directly: and method requests that cross an encapsulation boundary must do so via the boundary's owner object. The external object in Figure 4 cannot use the dashed red incoming link directly, but it could call a method on the Owner (dotted green line "1"). Being inside the encapsulation boundary, just like with owner-as-dominator, that method can then modify the internal object (dotted green line "2" in Figure 4). What makes owner-as-accessor different, is that the Owner object can both accept and return a direct reference to the Internal object (dashed red line). Such reference can be stored and passed around but cannot be used to modify the Internal object, unless the referrer is inside an appropriate ownership boundary.

Figure 5 shows an example following the one in Figure 4. The Internal object can have references to it stored outside its Owner (line 8), which would be illegal under owner-as-dominator. Any modification is only allowed via an appropriate owner (line 9) and not via external references (line 10).

Owner-as-accessor does not constrain the heap topology, but it does constrain the control-flow graph: a method invocation

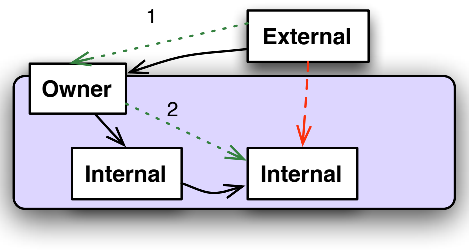

Fig. 4. References and Method calls

```
class Internal { String s = "abc"; }
class Owner {
 Internal i = new Internal();
 void modifyI() { i.s = "def"; } }
class External {
 Internal i; Owner o = new Owner();
 void doIt() {
  this.i = o.i; // OK. Stores external ref.
  o.modifyI(); // OK. Modifies Internal.
  // this.i.s = "ghi"; // WRONG. Not owner!
} }
```

Fig. 5. Owner-as-Accessor Code Example

upon an owner must dominate all method invocations upon all objects owned by that owner. This control-flow constraint is implicit in owner-as-dominator encapsulation: any design conforming to owner-as-dominator satisfies owner-as-accessor.

In this paper we are concerned with design changes, rather than the annotations required to express the properties of a design within an ownership type system, or the properties of the type system itself. Using the JavaCop program constraint system [16], we have implemented an ownership checker based on the theory of Tribal Ownership [17]. Each type used within a class is assigned to a single ownership mode. All instances of inner classes are owned by their enclosing objects, and additional classes can also be annotated as owned. Method invocations (for both owner-as-dominator and owner-as-accessor) and assignments (for owner-as-dominator) are checked to ensure they maintain the ownership invariants.

III. Collections and Ownership Case Study

The Java Collections Framework [18], [19] has formed an important part of the Java platform ever since its first release in JDK 1.2. Designed by Joshua Bloch, there are over 50 classes and interfaces in the framework. The core of the framework, however, are a relatively small number of interfaces to collection objects (Set, List, and Map), and a similarly small set of implementations of those interfaces. Our case study is based on the Java 5.0 version of Collections, as this is the version required by the DaCapo benchmarks, see Section IV-B.

The eight classes: ArrayList, LinkedList, HashMap, LinkedHashMap, TreeMap, HashSet, LinkedHashSet, TreeSet, plus the legacy classes Hashtable and Vector, form the backbone of the mainstream collections usage: our case study analyses these ten implementation classes. We consider the implementations of the two main abstractions — lists and maps — in turn, explaining their design; describing how (and how much) they encapsulate their representations; and if they do not, outlining refactorings to restore encapsulation. Our discussion focuses on lists and maps because HashSet and TreeSet are wrappers that implement sets using HashMap and TreeMap respectively. Our implementations are available[1].

A. Lists

Although the collection objects' interfaces are quite rich — ArrayList, for example, defines around thirty methods — for the purpose of this paper we need to consider only a few. Lists (and Maps) define get(index) and set(index,element) methods to read and write collection elements. Lists (and Sets) also define add(element) and remove(element) methods to add and remove elements. All collections support an iterator () method that returns a dependent Iterator object. An iterator supports at least next() and hasNext() methods that traverse through the collection an element at a time. We say the iterators are "dependent" on their underlying collection because they access (or even update) the actual elements stored in the collection.

1) ArrayList: An ArrayList is one of the simplest of the collections. An ArrayList<E> stores elements of type E in a primitive array, and copies and replaces that array as necessary as the collection grows and shrinks. Elements in an ArrayList are accessed by simply accessing the corresponding elements of the array, after checking they are within the range of valid elements:

Crucial to the correct operation of an ArrayList is that each underlying array is owned by the ArrayList whose elements it holds, and so must never be accessible from the outside. This is because if the list grows (or shrinks) the array will be replaced with a larger (or smaller) array, and the ArrayList elements copied from the old array into its new replacement.

This encapsulation is respected even by ArrayList's iterators. Implemented as an inner class[2], the iterator refers to its ArrayList via Java's implicit link between every inner class instance and their enclosing "outer" class instance. An ArrayList iterator maintains an integer cursor field that indexes the next element to be returned. The iterator's next() method simply returns the element at the cursor position, and then increments the cursor.

ArrayLists preserve encapsulation because their methods and iterators access their list only via that lists's public methods size() and get(). Even though it is implemented as an inner class, conceptually an ArrayList iterator is outside the ArrayList's encapsulation boundary — the iterator has no privileged access to its underlying ArrayList instance, and in particular does not access any private representation owned by the ArrayList (e.g. the underlying primitive array). An ArrayList object *always* acts as the single entry point for the list abstraction that it represents.

[1]http://www.ecs.vuw.ac.nz/~alex/software/files/own-coll-20120817.tgz
[2]Technically, an inner class of the AbstractList superclass of ArrayList, although that does not affect the encapsulation of the ArrayList.

Because ArrayList encapsulates its representation, the implementation does not need to change to satisfy owner-as-dominator encapsulation, and so also satisfies owner-as-accessor encapsulation. In this case at least, ownership does not impose any additional performance cost on the design.

2) Vector: The design of the legacy Vector class is broadly similar to ArrayList. The only issue is that the vector iterator (an instance of the legacy Enumeration interface) directly accesses the underlying array owned by the vector. We refactored this straightforwardly to use the public interface, as in ArrayList.

3) LinkedList: The other standard List implementation, is a more difficult case than ArrayList or Vector. As we discussed in the introduction, the LinkedList class maintains a doubly-linked list of link nodes (instances of the static inner class Entry<E>). The LinkedList's iterator maintains an incoming pointer (called next) that refers directly to the current link Entry. The iterator's next() method runs this pointer along the list's internal structure, returning the element at the current position. This structure sharing means that a LinkedList object is not a single entry point to its representation — every outstanding iterator on the LinkedList accesses the list Entry nodes directly.

4) Naïve Owner-as-Dominator LinkedList: Our first refactoring was simply to adopt the ArrayList iterator described above — the ArrayList iterator requires only a List interface, and LinkedList implements List. Unfortunately, we expected that this design would not perform very well and our initial tests confirmed our expectations. The problem is that the iterator calls get(index), and the get(index) method on a linked list must start from the beginning (or end) of the list and count along until it locates the list Entry holding the indexed element: An individual call to get() on a LinkedList will be $O(N)$ and a whole iteration will be $O(N^2)$. So: while adopting an ownership discipline can simplify the design of the LinkedList class — suggesting the use of a more abstract iterator — this naïve design would impose a substantial performance cost [3].

5) Single Place Cache: The JDK specification makes clear that programmers should expect the performance of LinkedLists will always be $O(N)$ for individual random accesses. The specification also makes clear that programmers should expect $O(N)$ for a full traversal via an iterator. The original iterators with direct pointers into the list delivered this performance. A slightly more complex design, however, can restore $O(N)$ traversals in most cases, while preserving owner-as-dominator encapsulation — the LinkedList object owns its list entries, and remains the single entry point of access to those list entries. We add a cache to the linked list that remembers the last accessed entry and its index. A call to get(index) can look into the cache, and update the cache once it has found the requested element.

In this design, a single (forward or reverse) traversal via an iterator, or even a traversal driven programmatically sometimes forwards and sometimes backwards, should have $O(1)$ performance for a single call to next() and consequently $O(N)$ for a full traversal of the list. The overhead of the cache itself should not be substantial. Random access to a LinkedList will still have $O(N)$ performance for each get(), but this is permitted by the framework specification, since LinkedList is not a random access structure. The specification also invalidates all except one iterator whenever the list is modified. This forbids modifications by more than one iterator, and so we expect a single place cache should suffice for most LinkedList use cases. Multiple simultaneous traversals that purely read data, however, would still revert back to $O(N^2)$ performance, but we expect such uses of LinkedLists to be rare, as ArrayLists are quicker to traverse than LinkedLists, and have around a quarter of the storage overhead.

[3]While it may seem unreasonable to replace an $O(1)$ operation with an $O(N)$ one, we have examined an average number of iterators per instance of a linked list in all of DaCapo benchmarks and found it to be low: 4 or 5 per list for the avrora and pmd and less than one per list for most others [20].

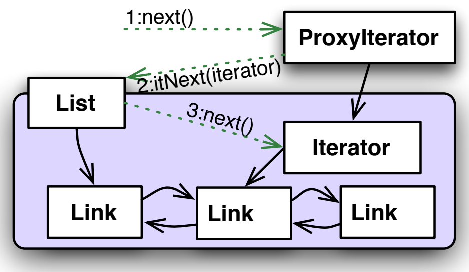

Fig. 6. List Iteration by Proxy

6) Owner-as-Accessor via External Proxy Iterator: As Figure 2 showed, the problem with the standard Java Iterator is that it is *outside* the List but must directly manipulate links. So long as the standard Iterator is only ever used *inside* the List, there are no encapsulation breaches. Figure 6 shows an alternative to this design that maintains owner-as-accessor encapsulation, but not owner-as-dominator.

A **proxy iterator** is a peer of the List, and is thus accessible outside. ProxyIterator maintains a reference to a standard Iterator inside the List, but does not use that reference directly. In response to a next() method invocation on the external proxy iterator, the proxy invokes itNext(iterator) on the List, passing the actual internal iterator as a parameter. At this point, control flow passes into the List object (thus maintaining owner-as-accessor) which then invokes next() on the encapsulated Iterator. The cost of this refactoring is creating the external proxy, plus redirecting its calls via the owning List. An important constraint of this design is that we must be careful to pass the proxy iterator only into the list to which it belongs. This can be checked statically by advanced ownership type systems [21], or it can be checked dynamically [15].

7) Owner-as-Accessor via Indirection Iterator: The proxy iterator refactoring avoids changing the standard list iterator, but requires creating the external proxy. Figure 7 shows our final list refactoring, using a single "indirection iterator" object that plays the roles of both external proxy and internal iterator.

The **indirection iterator** presents the standard Iterator interface (e.g. next()) and forwards those messages to its List, just like the external proxy. Whereas the external proxy passed the internal Iterator as an argument, here the indirection iterator

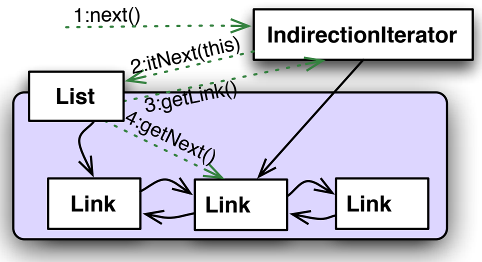

Fig. 7. List Indirection Iterator

passes *itself* as an argument. The implementation of itNext(itr) in List retrieves the current link (getLink) from the iterator and directly gets the next link (getNext). The external iterator keeps references directly into the List's internal links (rather than to a proxy) but never uses those links directly, thus maintaining owner-as-accessor encapsulation but not owner-as-dominator. We expect this refactoring to be more efficient than a proxy iterator, but it is more expensive to perform, as we cannot reuse the standard iterator object, but rather must incorporate its code into the itrNext()-style methods on the List object. And, as with the proxy design, we must ensure the indirection iterator is only passed into the list to which it belongs, checking either statically or dynamically.

8) Other Possible Iterator Refactorings: There are more alternatives to consider for ownership-aware list and iterator refactorings. The last author discussed the possibilities in detail in a separate paper [22]. For this presentation, we chose the ones we thought would be most representative and easy for software engineers to adopt in practice. More efficient versions include a "magic cookie iterator" [22] that stores an appropriate cached position for each existing iterator of the current list communicating using a simple unique id or "magic cookie".

B. Maps

The Map<K,V> interface is the other major interface within the collections framework. Maps provide put(K,V) and V get(K) methods to store and retrieve values V associated with keys K. Maps are not directly iterable — rather they provide methods that return three separate dependent iterable views: Set<K> keySet(); Collection<V> values() and Set<Map.Entry<K,V>> entrySet(), all of which have their own iterators that support modification. The key set (and value collection) contain a set of all the keys (and a bag of all the values) in the underlying map: the entry set is a set of objects that each represent a single key-to-value association and implement the Map.Entry<K,V> interface.

The key encapsulation issue with the core Map implementations is that the Entry objects available via the entry set are the very same entry objects that implement the map. Furthermore, the key and value sets are implemented in terms of the entry set and the entries it contains. This is a tightly coupled design: Figure 8 attempts to show these interrelationships.

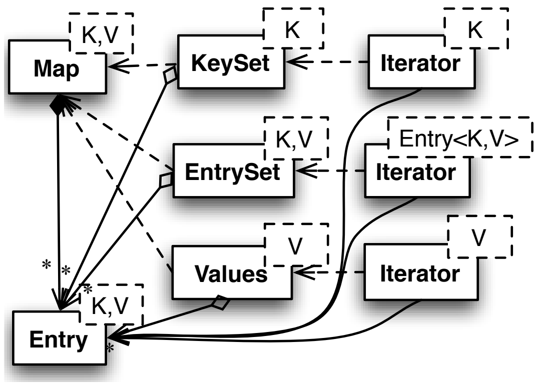

Fig. 8. Map Interfaces. Dashed lines show conceptual dependencies, while solid lines show references in most implementations.

1) Refactoring HashMap: For this case study, we first aimed to find a design for maps with a single point of entry that could maintain owner-as-dominator encapsulation over as much of the map as possible. In particular, the entry objects storing the key-value mappings need to be protected from outside access. This is straightforward. The problem then is that our refactored maps need to preserve the existing collections' Map interface, so we could exchange implementations to benchmark each design. This means that the dependent set views of a map (the key set, values collection, and entry set); their iterators; and the entry objects must remain available to Map clients.

To support these use cases, we again introduce proxies for these objects as peers of the maps as shown in Figure 9. Writes to the proxies update the underlying map by calling put(K,V) through the main map object's interface, rather than by being part of the map's representation themselves. Within the body of a map implementation (say HashMap) there are three Entry types: Map.Entry is the common public interface used by clients, HashMap.Entry is the owned inner class; and EntryProxy is the (common) peer class. The ownership discipline ensures the peer entry proxies cannot be confused with the owned entry objects and vice-versa.

As a result, the Entry<K,V> objects are not accessible outside of the Map<K,V> object that owns them. All three of the map views (key and entry sets and values collection) and their respective iterators *do not have direct references to map's entries*. Instead they work with EntryProxy<K,V> objects that mirror the Entry<K,V> objects encapsulated inside the map and store the same key-value pair as the mirrored entry without allowing direct access to the internal structure of the map. Any modification to such entry proxies will not have any effect on the map's internal representation and any code that relies on modifying the map by manipulating is entries directly as opposed to using map's public interface will no longer work. An important result of our study is that in none of the benchmarks under consideration have we found any instances of such manipulation and thus the only reason for exposing the internal entries behind each map is presumed efficiency improvement.

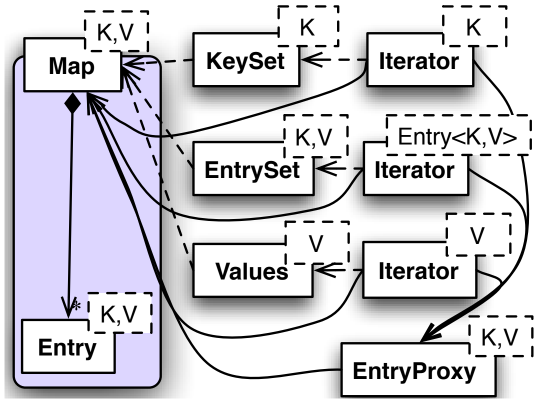

Fig. 9. Refactored Map. The Map's Entry instances are owned by the Map object. Dependent views and iterators access the Map only via the Map object's interface, and substitute EntryProxy objects for Entry objects.

The dependent view iterators are the most challenging part of this refactoring. All three iterators are subclasses of the abstract inner class HashIterator<E> which traverses the HashMap by accessing the underlying table and Entry objects directly. To restore owner-as-dominator encapsulation, we refactored the design to introduce a completely external iterator, similar in style to the ArrayList iterator, that does not use incoming pointers into the HashMap's implementation. This iterator keeps track of the current and next keys, and uses new K getFirstKey(), K getNextKey(K key), and hasNextKey(K key) methods on the HashMap to manage the iteration. The implementation specific code from the original iterators' nextEntry() method is refactored to call the map's getNextKey() method.

2) LinkedHashMap: The LinkedHashMap class extends the HashMap by threading a doubly-linked list between the table Entry objects to support quick and stable traversal. Once we had refactored HashMap, the only change LinkedHashMap required was to ensure the getNextKey() method traversed the list.

3) Hashtable: The design of the legacy Hashtable class is broadly similar to HashMap, with the same underlying design from Figure 8 and the same epidemic of inner classes and aliases, although the method names and interfaces are not compatible. Like the other legacy class, Vector, all accesses to a Hashtable are synchronized. We were able to encapsulate Hashtable using very similar refactorings to HashMap. Our refactored implementation also reuses the common Map key-based traversal methods and the common EntryProxy class. We had to refactor the existing Hashtable iterators, rather than reuse the shared implementation, because Hashtables must support both the Iterator and the legacy Enumeration interfaces.

4) TreeMap: We originally planned to refactor the TreeMap class separately, once we had completed the refactorings of the various hash-based maps. Considering the refactored map design (Figure 9) we realised that we would in fact be able to re-use all the iterator, view, and entry objects from HashMap, because they can only communicate with their underlying map via that map's public interface: encapsulating the representation has also abstracted the representation behind that interface (albeit extended with the various getFirstKey() / getNextKey(K key) methods). In the same way as the residual public interface of a linked list is the List interface, the residual public interface of the various map classes is just this extended Map interface, and so one single, reusable, common external iterator class suffices to iterate over any kind of map.

Again, we expect that the narrower interface between an iterator and its underlying collection will decrease the efficiency of the iteration. Calling getNextKey(currentKey) will require tracing down from the root of the Red-Black tree to the node holding that key, and this is certainly more work than just following a pointer directly. We also tried single place caches (as with LinkedLists, Section III-A5), but disabled them for our tests as the DaCapo xalan benchmark did not tolerate the resulting behaviour.

5) Owner-as-Accessor Maps: Finally, we refactored all the maps to maintain owner-as-accessor encapsulation rather than owner-as-dominator. We applied the two refactorings we used with LinkedList to each Map, in spite of the differences between Map implementations. We built external proxies for the various iterators, sets, and entries that simply stored references to the standard iterators etc., which were treated as internal to the maps, just as in the LinkedList proxy iterator (Section III-A6 and Figure 6). These proxies were able to be reused across all the Map implementations. Then we built indirection iterators, sets, and entries following the design of the corresponding indirection objects for the LinkedList (Section III-A7).

IV. Experimental Methodology

To evaluate our refactorings, we performed a number of benchmark studies comparing the original and refactored collections: we include three small microbenchmarks and the three major benchmark suites: DaCapo, SPECjbb2005 and SPECjvm2008. The refactored collections include: **OasD** that implements owners-as-dominators discipline, **Proxy** that implements owners-as-accessors using a proxy iterator, **Proxy Dynamic** that also performs a run-time ownership check, **Indirection** that implements owners-as-accessors using an Indirection Iterator, and **Indirection Dynamic** that also performs a run-time ownership check. In this section we describe our methodology, and present the results in the next section.

A. Microbenchmarks

These three microbenchmarks test collections' performance, focusing on the cost of iterating over a whole collection.

1) The IteratorLoops test from the Doug Lea's JSR166 collections microbenchmarks [23]. This runs a large number of traversals over partially filled collections with occasional additions of elements. The result is the time taken for a single `next()` step of an iterator.
2) LinkedList iteration: (a) forwards; (b) backwards; (c) forwards, but with two iterators interleaved, the second iterator indices after the first. The last test disrupts the caching algorithm in the refactored linked list.
3) The MapMicroBenchmark test from the Doug Lea's JSR166 collections microbenchmarks [23]. This runs a

large number of element-level map operations reflecting a typical usage of maps in the real world and reports an average time an operation takes. This works on on table-based map implementations only (i.e. not TreeMap) and we evaluated different map sizes in increasing order.

To make sure that our numbers were not disturbed by the garbage collector or just-in-time compiler, we warmed up the VM before timing the tests. We set the –XX+PrintGC and –XX+PrintCompilation options and checked our traces that compilation or collection did not occur during timed runs. Each microbenchmark was run 25 times and the results analysed.

B. DaCapo Benchmarks

The DaCapo benchmark suite [24] is a well-established benchmark suite *representative of typical Java loads*. We included every benchmark in DaCapo in our study: avrora, batik, eclipse, fop, h2, jython, luindex, lusearch, pmd, sunflow, tomcat, tradebeans, tradesoap, and xalan. The DaCapo benchmarks come with data sets of different sizes. We used the *large* size for each benchmark where it was available. The fop and luindex do not include a large size, so for these two benchmarks we used the *default* size.

We carried out 5 runs of 30 iterations of each benchmark, of which the last 5 are used, resulting in 25 data points for each benchmark in each condition [25].

C. SPECjbb2005 Benchmark

SPECjbb2005 is a Java Server Benchmark capturing the common types of server side Java applications today. SPECjbb2005 is well-known for making a heavy use of collections and was thus considered essential to be included in our study. Following the other benchmarks in our paper, we ran SPECjbb2005 on a default set of 16 warehouses 25 times and report the averages of these runs for different collections implementations.

D. SPECjvm2008 Benchmark

SPECjvm2008 is a Java Virtual Machine Benchmark that measures the performance of a typical Java Runtime Environment using a selection of real life applications focusing on core Java functionality. We included it for completeness as a more traditional macrobenchmark with a caveat that DaCapo was designed to improve on the number of shortcomings of SPEC-style benchmarking [24].

E. Execution Environment

Our choice of Java Development Kit (JDK) version and thus Java Collections Framework implementation was driven by the latest version of DaCapo Benchmark Suite [24]: v9.12, released in December 2009. DaCapo was built using Java v1.5.0 and this is the version of JDK and Java Collections Framework that we used: in particular the 1.5.0_22 version as the latest Java v1.5.0 version available on the Oracle web site. We executed all the tests on the Java HotSpot(TM) Server VM (build 1.5.0_22-b03, mixed mode). We compiled the modified collection implementations with javac version v1.5.0_22. This meant that we had to omit one of the SPECjvm2008 benchmarks (xml.transform) that requires a later version of Java.

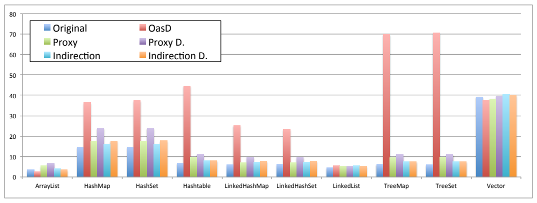

Fig. 10. Microbenchmark 1: IteratorLoops (single iteration time, *ns*)

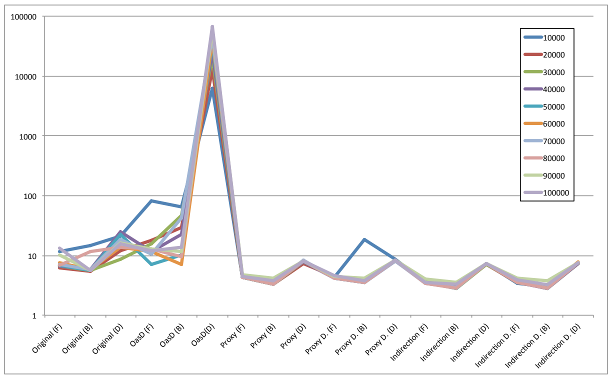

Fig. 11. Microbenchmark 2: Linked List Single Iteration, *ns*, log-scale

We placed the modified collections and the same classes that were unmodified in the “ownership” and “original” folders and then utilised the Java –Xbootstrappath/p: option to place our classes in the beginning of the JVM boot class path.

We ran all our tests on a Ubuntu Linux v11.4 machine using v2.6.38-8 SMP kernel with the SMP option selected in the kernel, configured to only use 7 of the 8 cores available in our Dell OptiPlex 790 (Intel Core 2 i7-2600 CPU 3.40 GHz with 4GB of RAM). We used the Linux taskset command to set the CPU affinity of our Java Virtual Machine to the unused core to minimise disturbance from the rest of the operating system.

V. Results

A. Microbenchmarks

1) IteratorLoops: Figure 10 shows the results of the IteratorLoops benchmark from JSR166 benchmarks for all the collections in the *original* and five *ownership refactored* variants. The figure plots the mean time (in nanoseconds) for 1 iteration step (i.e. a call to next).

We can observe that owner-as-dominator refactored implementations were slower per iteration by a factor of 3, except for Hashtable and TreeMap (and hence TreeSet), which are slower by factors of seven or eight. In the case of Hashtable the fact that every method is synchronized, including helper methods to get next key or check for modifications, seems to have played a major part. In the case of the tree-based collections, it is indeed much slower to search for the next entry from the root of the tree, rather than following an incoming pointer.

However, observe that all four of the owner-as-accessor refactored implementations were not significantly different from the original collections.

TABLE I
FULL RESULTS TABLE (FORMAT: MEAN|SD; STATISTICALLY SIGNIFICANTLY DIFFERENT VALUES SHOWN IN BOLD)

Benchmark	Original		OasD		Proxy		Proxy D.		Indirection		Indirection D.		Number	Percent
DaCapo (Time in *ms*; lower is better)														
avrora	23003	300	**22781**	**415**	**22816**	**236**	22867	179	22948	389	22873	323	219049309	78.08%
batik	2516	34	2517	19	2520	25	2519	19	2519	25	**2528**	**30**	26507124	31.37%
eclipse	53793	1031	53480	936	53716	1010	53812	1329	**53554**	**1406**	53608	738	355465429	33.63%
fop	393	27	397	29	394	20	397	23	396	23	399	20	1874892	18.07%
h2	24133	580	24238	593	24141	517	24188	380	23967	320	23934	375	90175446	8.04%
jython	15041	215	**15476**	**100**	**15725**	**107**	**15719**	**53**	**15837**	**110**	**17050**	**145**	159700109	7.49%
luindex	705	18	**687**	**23**	714	22	713	19	710	40	718	51	327466	36.66%
lusearch	7251	184	7334	105	**7181**	**189**	**7120**	**198**	7226	299	7325	78	11979688	5.45%
pmd	3944	47	**3992**	**39**	**4046**	**59**	**4065**	**72**	**4005**	**64**	**4054**	**67**	10712544	36.55%
sunflow	22656	518	22560	365	**23365**	**126**	22970	711	22851	523	**22331**	**207**	171198077	<0.01%
tomcat	7576	108	7641	135	**7687**	**134**	**7736**	**111**	**7733**	**88**	**7661**	**118**	16726923	13.95%
tradebeans	27952	409	**27556**	**494**	**28258**	**506**	28142	275	27998	328	28020	340	1621619	33.00%
tradesoap	64476	1251	65193	1549	65042	1712	65119	1463	64390	1378	65111	1499	1631193	32.82%
xalan	26604	384	26692	318	**26383**	**247**	**26173**	**258**	**26125**	**251**	**26310**	**291**	61153799	13.23%
SPECjbb2005 (Throughput; higher is better)														
SPECjbb2005	29598	405	**14062**	**181**	**28825**	**860**	**28540**	**619**	**28959**	**764**	**28394**	**641**	35542855	4.87%
SPECjvm2008 (Time in *ms*; lower is better)														
compress	46.56	0.81	46.77	0.59	46.71	0.59	46.82	0.42	46.83	0.42	46.84	0.57	199478	20.21%
crypto.aes	18.49	0.18	18.40	0.13	18.46	0.10	18.48	0.18	**18.42**	**0.10**	18.48	0.19	254853	22.00%
crypto.rsa	35.04	0.25	34.89	0.28	34.92	0.31	34.95	0.36	34.94	0.25	34.95	0.27	6535358	11.18%
crypto.signverify	53.06	0.27	52.97	0.31	53.11	0.29	53.16	0.36	53.09	0.38	53.12	0.29	3290783	2.13%
derby	21.55	0.48	21.63	0.40	21.56	0.50	21.73	0.43	21.59	0.38	21.61	0.33	89061937	3.04%
mpegaudio	15.08	0.05	15.10	0.05	15.08	0.06	15.11	0.05	15.10	0.06	15.07	0.06	209089	20.32%
fft.large	23.61	0.27	23.53	0.30	23.49	0.27	**23.43**	**0.28**	23.55	0.24	23.51	0.38	168290	23.78%
fft.small	82.75	4.49	84.24	5.21	84.98	3.94	83.35	4.01	84.46	4.70	83.57	4.26	439646	9.22%
lu.large	6.98	1.44	6.69	1.24	6.72	1.22	7.05	1.39	6.34	0.96	7.02	1.42	166728	23.92%
lu.small	107.98	0.87	107.61	0.92	107.48	0.75	107.58	0.84	107.94	0.70	107.82	0.85	642713	6.39%
monte_carlo	15.33	1.49	15.33	1.48	15.65	0.10	15.67	0.11	15.63	0.03	15.29	1.47	179253	22.58%
sor.large	13.01	0.02	13.01	0.02	13.01	0.01	13.02	0.02	**12.99**	**0.05**	13.01	0.01	167449	23.92%
sor.small	57.47	0.11	57.47	0.10	57.49	0.10	57.47	0.12	57.43	0.07	57.44	0.10	193425	21.16%
sparse.large	11.51	0.23	**11.97**	**1.22**	11.70	0.28	11.82	0.80	11.71	0.34	11.77	0.36	166691	23.98%
sparse.small	43.87	0.11	43.79	0.12	43.80	0.18	43.83	0.11	43.85	0.18	43.80	0.15	182676	22.17%
serial	33.22	0.98	**32.46**	**0.96**	33.27	0.81	33.04	0.92	32.88	1.31	33.31	0.89	51123977	5.68%
sunflow	20.51	0.50	20.48	0.50	20.55	0.32	20.42	0.52	**20.22**	**0.63**	20.51	0.47	42506559	0.12%
xml.validation	63.38	1.09	63.01	0.96	63.36	1.03	63.19	1.42	63.75	1.39	63.23	1.03	10318760	3.33%

2) List Iteration: Figure 11 presents a microbenchmark comparing the original and refactored versions of the linked list: note the log scale on the y-axis. The graph shows the time for a single next call. The "disruptive" benchmark for owner-as-dominator version shows linear performance for a single step, (thus $O(N^2)$ overall) while the other iterators (including all owner-as-accessor implementations) behave linearly for different collection sizes (i.e. $O(N)$ overall as expected). As collection sizes get larger, the amortised time for the refactored collections approaches that of the original collections.

3) Map Iteration: Figure 12 presents results of the Map-MicroBenchmark benchmark from JSR166 benchmarks for three different kinds of maps: HashMap, LinkedHashMap, and Hashtable. We used the default parameters for the microbenchmark and map sizes and report the averaged results for the largest map size. Observe that owner-as-dominator performs only 20% slower than the original while all four of the owner-as-accessor perform with no more than 3-5% slowdown.

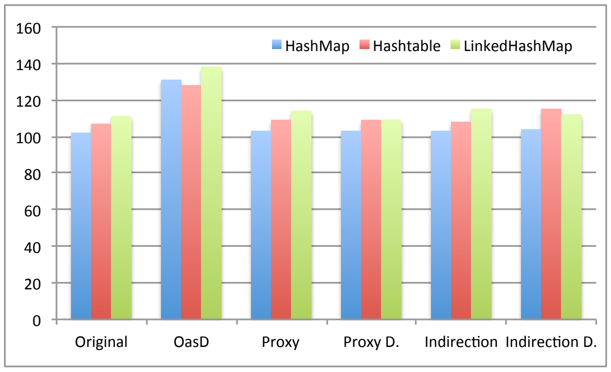

Fig. 12. Microbenchmark 3: MapMicroBenchmark (*nano*seconds per element operation (averaged across get, put etc) across different element types)

B. DaCapo, SPECjbb2005, and SPECjvm2008

Table I presents the results of the DaCapo, SPECjbb2005, and SPECjvm2008 benchmarks. displays the mean runtimes. For each benchmark, for standard and refactored collections, the time taken was recorded for 25 runs. We wanted to test whether the mean time taken differed between the original and five ownership versions, while controlling for the different times needed for different benchmarks. We show the average times across all runs and include a standard deviation for all DaCapo benchmarks. For SPEC benchmarks we report a throughput in either "bops" or "operations per minute" and again include the standard deviation alongside the mean across 25 runs.

Usually one would use Analysis of Variance to test the null hypothesis that the mean time taken did not differ between the standard and refactored collections. However, ANOVA assumes the data are normally distributed and that the variances of time taken are the same in all $2 \times 14 = 28$ groups. In our data, these assumptions did not hold for all benchmarks. Data transformation did not solve this problem. We therefore used non-parametric methods (which do not require normality nor equality of variances) – in particular the Asymptotic p-value of the Mann-Whitney U test – separately for each benchmark, to determine whether we could reject the null hypothesis that the mean time taken for the standard and refactored collections was the same. To perform these tests, we used PASW Statistics 18 Release 18.0.0 (Jul 30 2000), hosted by Microsoft Windows Server 2003 Standard Edition Service Pack 2, running on an Intel Xeon 5130 2.00GHz with 4GB of RAM.

Table I includes the results of the Mann-Whitney test, showing the refactored results with a statistically significant difference from the performance of the original collections in bold. We reject a null hypothesis at significance level $p < 0.05$. We can reject the null hypotheses that the means are the same for the refactored implementations for 40 of the 165 refactored benchmarks: for the others, we were unable to show a statistically significant difference. We refer the interested reader to a technical report accompanying this paper [20] that contains the obtained p-values and clustering results.

Finally, we also measured the number of objects instantiated during each benchmark run and the percentage of them that were collections from the java. util package [26]. Having observed a significant slowdown for the SPECjbb2005 in the case of owner-as-dominator, we have also measured the time this benchmark spent in the methods of java. util collections as the percentage of objects was low (4.87%). We used −Xrunhprof:cpu=times to obtain such timings and found that the original version spent 7.79% of its time in the collections methods and the owner-as-dominator spent 13.94% of its time in the collections methods (almost all of it in TreeMap). We hypothesise that the lack of caching used in our owner-as-dominator map implementations caused this and we can see a number of ways in which this can be improved similar to the way linked lists can be made faster with caching.

VI. Discussion and Related Work

The results we have presented have at least two alternative interpretations. First, incoming aliases into collection implementations are absolutely necessary in specific cases, as refactoring to an encapsulated interface means collection operations' runtime performance will be five to ten times slower than otherwise. Second, incoming aliases into collection implementations are clearly unnecessary in general, as the largest significant slowdown in the DaCapo experiment was less than 2%. The truth, no doubt, lies somewhere in-between.

The reasons for the microbenchmark results, at least, seem clear: more general (and reusable) interfaces are also by necessity less efficient. Our refactorings imposed additional hash lookups, and additional list and tree traversals, when an incoming pointer could take the program directly to exactly the right place without any such overhead.

The reasons for the macrobenchmark results are less clear. Perhaps there is a small effect, but the variability introduced by a JITting VM, garbage collector, the underlying operating system mean the effect is lost in the noise. Arguably, however, most programs would not be affected by such a small overhead. Perhaps collections do not make up a significant portion of the DaCapo benchmarks execution time? This is certainly the case for sunflow and luindex, although as Table I illustrates, most of the benchmarks create tens of thousands of collection objects, and some benchmarks create millions. The DaCapo benchmarks have been selected to model realistic Java workloads [24]: if they are a reasonably accurate gauge of the use of collections in Java programs, then we would not expect significantly higher runtime impacts upon other Java applications.

a) Evaluating Ownership: There have been a number of implementation studies evaluating ownership types [27], [11] — some quite extensive [28]. Generally these studies were undertaken in the context of validating a particular type system proposal and the performance evaluation did not specifically concentrate on the costs of various ownership-friendly designs. Many of these studies used the collections library as an example, and were able to check the whole of the collections library, albeit with varying amounts of annotation, depending on the system. The AJ collections reimplementation in particular included some performance analysis using a selection of Java applications and SPECjbb benchmark — they found that a tuned version of AJ collections performed only marginally slower than the standard Java version (Figure 22 in [11], which aligns well with our findings.

These studies differ from the approach we have adopted here because they were mostly based on more flexible (i.e. less encapsulating) ownership disciplines — confining objects within regions rather than per-object ownership [29], or by permitting incoming pointers in some circumstances [27]. The problem is that these supposedly "benign" incoming references can drastically reduce encapsulation.

There have been surprisingly few stand-alone case studies evaluating ownership *per se*. The closest research to this work is Stefan Nägeli who studied how ownership affected design patterns and the Swing GUI library [30], and Cele and Stureborg [31] who implemented three medium-sized programs while respecting an strong ownership discipline. Both studies found that ownership could help structure programs, but could also be a cause of refactoring and redesign: this chimes with our experience. Neither study considered the potential performance cost of the encapsulation enforced by ownership.

b) Inversion of Control: Many of the problems we encountered, particularly with the Map iterators, can be understood in terms of inversion of control: we would have had no problem writing an "internal iterator" method for each collection implementation to iterate over all their elements. Because these methods are encapsulated within each collection class, there is no need (or temptation) to breach encapsulation. The reason, of course, is that in an internal iterator, control and data flow are both efferent, flowing from the collection to the iterator . By contrast, an in external iterator, data flow remains efferent, but control flow is afferent: the external iterator calls in to the collection that hands each element back in turn.

CLU-style generators [32] (as popularised in Python, Ruby, and C♯) are an alternative solution to this problem: programs are written as if they used simple internal iterators — with all the benefits for encapsulation that implies — and then the generator construct inverts the control flow.

c) Further Work: The Java collections continue to evolve. We worked with the version 5 collections because that worked with DaCapo. Repeating this study once closures support is integrated into the main Collection APIs (and once benchmarks have evolved to rely upon those APIs) could address some of the hypotheses above regarding inversion of control.

VII. Conclusion

In this paper we present the first experimental evaluation of the cost of ownership types. We examined the use and breaches of encapsulation in the core classes of the Java Collections Framework. We refactored those classes as necessary to fit the owner-as-dominator and owner-as-accessor encapsulation disciplines. We measured the overhead of these refactorings that showed encapsulation reduces iteration performance by factors of 2 to 8. Finally, we compared the performance of the DaCapo, SPECjbb, and SPECjvm benchmark suites, gaining statistically significant results for a number of benchmarks, SPECjbb owner-as-dominator demonstrating the largest slowdown. Owner-as-accessor slowed down no more than 3% for all benchmarks, even with dynamic ownership checking.

We hope these results may encourage object-oriented designers to consider object encapsulation more carefully when designing their programs — especially their use of incoming aliases to circumvent encapsulation — and to ask themselves: are their incoming aliases really necessary?

Acknowledgments

Thanks to Dalice Sim, VUW Statistical Consultant, for the statistical analysis; to Doug Lea, Tony Hosking, and Andy Georges, for help with DaCapo and Collections; to Shane Markstrum for help with JavaCOP; to Christo Fogelberg and Andrew Walbran for work on the encapsulated collections; and to the anonymous reviewers. Funded by RSNZ Marsden Fund.

References

[1] D. Clarke, J. Potter, and J. Noble, "Ownership types for flexible alias protection," in *OOPSLA*. ACM Press, Oct. 1998, pp. 48–64.

[2] J. Potter, J. Noble, and D. Clarke, "The ins and outs of objects," in *Australian Software Engineering Conference*. Adelaide, Australia: IEEE Press, November 1998, pp. 80–89.

[3] T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek, "Implicit ownership types for memory management," *Science of Computer Programming*, vol. 71, no. 3, pp. 213–241, 2008.

[4] D. Clarke, M. Richmond, and J. Noble, "Saving the World from Bad Beans: Deployment-Time Confinement Checking," in *OOPSLA*. Anaheim, CA: ACM Press, 2003, pp. 374–387.

[5] W. Dietl and P. Müller, "Universes: Lightweight ownership for JML," *Journal of Object Technology*, vol. 4, no. 8, pp. 5–32, 2005.

[6] S. Srinivasan and A. Mycroft, "Kilim: Isolation-typed actors for Java," in *ECOOP*. Springer-Verlag, 2008, pp. 104–128.

[7] B. Bloom, J. Field, N. Nystrom, J. stlund, G. Richards, R. Strnisa, J. Vitek, and T. Wrigstad, "Thorn: robust, concurrent, extensible scripting on the JVM," in *OOPSLA*, 2009, pp. 117–136.

[8] P. Haller and M. Odersky, "Capabilities for uniqueness and borrowing," in *ECOOP*, 2010, pp. 354–378.

[9] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, "Streamflex: high-throughput stream programming in Java," in *OOPSLA*. ACM Press, 2007, pp. 211–228.

[10] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A Type and Effect System for Deterministic Parallel Java," in *OOPSLA*. ACM Press, 2009, pp. 97–116.

[11] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek, "A data-centric approach to synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 1, p. 4, 2012.

[12] J. Noble, J. Vitek, and J. Potter, "Flexible Alias Protection," in *ECOOP*, vol. 1445. Springer-Verlag, Jul. 1998, pp. 158–185.

[13] J. Aldrich and C. Chambers, "Ownership domains: Separating aliasing policy from mechanism," in *ECOOP*, vol. 3086. Oslo, Norway: Springer-Verlag, Jun. 2004, pp. 1–25.

[14] P. Müller and A. Poetzsh-Heffter, "Universes: a Type System for Controlling Representation Exposure," Fernuniversität Hagen, Tech. Rep. 263, 1999.

[15] D. Gordon and J. Noble, "Dynamic ownership in a dynamic language," in *DLS Proceedings*, 2007, pp. 9–16.

[16] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble, "JavaCOP: Declarative pluggable types for Java," *TOPLAS*, vol. 32, no. 2, pp. 4:1–4:37, 2010.

[17] N. R. Cameron, J. Noble, and T. Wrigstad, "Tribal ownership," in *OOPSLA*, 2010, pp. 618–633.

[18] M. Naftalin and P. Wadler, *Java Generics and Collections*. O'Reilly Media, Inc., 2006.

[19] J. Bloch, *The Java Tutorials — Collections*, Oracle Corporation, 2011, http://download.oracle.com/javase/tutorial/collections/.

[20] A. Potanin, M. Damitio, and J. Noble, "Ownership and collections: The statistical analysis," ECS, VUW, http://ecs.victoria.ac.nz/Main/TechnicalReportSeries, Tech. Rep. ECSTR12-22, 2012.

[21] J. Östlund and T. Wrigstad, "Multiple aggregate entry points for ownership types," in *ECOOP*, LNCS, vol. 7313, 2012, pp. 156–180.

[22] J. Noble, "Iterators and encapsulation," in *TOOLS Europe*. IEEE Press, 2000, pp. 431–442.

[23] D. Lea, "Jsr166 loops benchmarks," http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/test/loops/, 2012.

[24] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*, Oct. 2006, pp. 169–190.

[25] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *OOPSLA*. Montreal, Quebec, Canada: ACM Press, Oct. 2007, pp. 57–76.

[26] S. Nelson, D. J. Pearce, and J. Noble, "Profiling object initialization for java," in *Third International Conference on Runtime Verification*, ser. RV 2012. Springer-Verlag, September 2012, pp. 292–307.

[27] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst, "Ownership and Immutability in Generic Java," in *OOPSLA*. ACM, 2010, pp. 598–617.

[28] J. Aldrich, V. Kostadinov, and C. Chambers, "Alias Annotations for Program Understanding," in *OOPSLA*. ACM Press, 2002, pp. 311–330.

[29] C. Grothoff, J. Palsberg, and J. Vitek, "Encapsulating Objects with Confined Types," *TOPLAS*, vol. 29, no. 6, 2007.

[30] S. Nägeli, "Ownership in design patterns," Master's thesis, Software Component Technology Group, Department of CS, ETH Zurich, 2006.

[31] G. Cele and S. Stureborg, "Ownership types in practice," Master's thesis, Department of CSS, Stockholm University, Jan. 2005.

[32] B. Liskov and J. V. Guttag, *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, 1986.