Implementation and Use of Data Structures in Java Programs

Syed S. Albiz and Patrick Lam
University of Waterloo

ABSTRACT

Programs manipulate data. For many classes of programs, this data is organized into data structures. Java's standard libraries include robust, general-purpose data structure implementations; however, standard implementations may not meet developers' needs, forcing them to implement ad-hoc data structures. We investigate the implementation and use of data structures in practice by developing a tool to statically analyze Java libraries and applications. Our `DSFinder` tool reports 1) the number of likely and possible data structure implementations in a program and 2) characteristics of the program's uses of data structures. We applied our tool to 62 open-source Java programs and manually classified possible data structures. We found that 1) developers overwhelmingly used Java data structures over ad-hoc data structures; 2) applications and libraries confine data structure implementation code to small portions of a software project; and 3) the number of ad-hoc data structures correlates with the number of classes in both applications and libraries, with approximately 0.020 data structures per class.

1. INTRODUCTION

> "In fact, these days, if I catch a programmer writing a linked list, that person had better have a very good reason for doing so instead of using an implementation provided by a system library."
>
> (Henning, [9])

Data structures are a fundamental building block for many software systems: computers manipulate information, and this information is often stored in data structures. Implementing linked data structures invariably appears early in an undergraduate Computer Science curriculum. Classically, programmers implement in-memory data structures with pointers.

Modern programming environments, however, include rich standard libraries. Since version 1.0, Java has included data structure implementations in its library. Java 2's Collections API [22] defines standard interfaces for data structures and includes implementations of standard data structures. While the general contract of a data structure is to implement a mutable set, general-purpose data structure implementations might not meet developers' specific needs, forcing them to implement ad-hoc data structures.

The goal of our research is to empirically investigate data structure implementation and use in Java programs. We are particularly interested in how programs organize information in the heap: do they use system collections such as `LinkedList` and `HashMap`, or do they implement their own ad-hoc lists, trees, graphs, and maps using unbounded-size pointer structures, like C programmers? Our results can help guide research in higher-level program understanding and verification (e.g. [2, 11]) and the development of software maintenance tools by identifying the code idioms that analysis tools need to understand. For instance, linked-list data structure manipulations require shape analysis techniques. Our operational definition of a data structure is therefore driven by static analysis considerations: what types of analysis suffice to understand typical Java applications? However, while our primary motivation is to investigate the necessity for shape analysis, we believe that our results have broader implications to software engineering in general, especially in terms of understanding how programs are built.

In this paper, we present the results of our analysis of data structure implementation and use in a corpus of 62 open-source Java programs and libraries. We identified a number of key features common to heap data structure implementations and implemented the publicly-available `DSFinder` tool. Our tool accepts a Java program as input and emits summary and detailed information about data structure and array uses and implementations in that program. We formulated and tested a number of hypotheses about data structures on our corpus. We found that Java programs rarely implement data structures—no benchmark implemented more than 24 linked data structures. As expected, our benchmarks extensively used the Java Collections. The number of data structures was correlated with the size of the program for both Java applications and libraries. We also found that data structure implementations were confined to small portions of programs' source code, as one might expect for maintainability reasons.

Our analysis tool identifies data structure implementations by searching for recursive type definitions and arrays, which signal the possible presence of sets of unbounded size. A simple analysis of a Java program's class definitions (available in the program's bytecode) thus suffices to identify its potential data structures. Our tool applies several automatic type- and name-based classification steps to the set of potential data structures and outputs this set. We manually investigated each potential data structure and classified it as a graph, tree, or list. Our tool also emits information about static counts of data structure usage, namely instantiations and field declarations, as well as counts of array usage—arrays are an alternative to collections.

Many classes of Java programs (such as web applications) are tightly coupled to databases. Databases provide an alternative to data structures, as they can store (persistent) data. However, since database use is typically costly and often involves interprocess or network communication, databases are typically used for persistent storage, and commonly-accessed data remains in the heap.

Implications.

Beyond contributing to understanding how Java software systems are actually built in practice—a valuable contribution in itself—our research has many implications to static analysis and software engineering.

- **Static Analysis.** A substantial body of literature (for instance, [3, 8, 16, 20]) contributes techniques for statically understanding the behaviour of programs that manipulate linked data structures. These shape analysis techniques can verify linked list and other data structure manipulations, including insertions, removals, and even list reversals. However, because shape analysis techniques are prohibitively expensive to apply to large programs, researchers have developed ways to mitigate the cost of these techniques.

  If data structure implementations are rare, then it is reasonable to expend significant effort (in terms of both annotation burden and analysis time) to successfully analyze the few data structure implementations. Analysis tools can then proceed to the verification of higher-level program properties, assuming that the data structure implementations have successfully been verified, using much more scalable static analysis techniques for large sections of program code.

- **Program understanding.** Some program understanding tools help developers understand how programs behave around data structures. For instance, Lackwit [17] infers extended types for C programs that help developers understand how programs manipulate abstract data types. Also, when verifying software models (e.g. a Flash filesystem [10]), the sets from the models must map to the data structures in the software. The Unified Modelling Language contains collections as a primitive. These techniques all rely on understanding a program's data structures.

- **Parallelization.** Much of the existing pointer analysis research sought to enable automatic parallellization: tree traversals, in particular, are particularly easy to parallelize. While our techniques do not automatically identify tree data structures, our manual analysis of the data gives insight into the question of how often trees occur in practice.

- **Library sufficiency.** Our analysis tool can identify areas in which the standard library is lacking: if we find that developers often implement a particular kind of data structure, then such a data structure ought to be added to the standard library. Note also that our analysis could identify instances where a feature already exists in the standard library, but that this feature is not sufficiently well-documented.

Our paper makes the following contributions:

- We propose the concepts of identifying possible data structures by type declarations and classifying probable data structures using field and type information.

- We implement the `DSFinder` tool, which reads Java bytecode and outputs information about data structure use.

- We collect a substantial corpus of open-source Java applications and apply our tool to this corpus.

- We formulate and empirically verify a number of hypotheses about how programs implement and use data structures; our corpus typically implements about 0.020 data structures per benchmark class.

```
public class LinkedList<E>
      extends AbstractSequentialList<E>
      implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    private transient Entry<E> header = new Entry<E>(null, null, null); // etc

    private static class Entry<E> {
        E element;
        Entry<E> next;
        Entry<E> previous; // etc
```

Figure 1: `LinkedList` implementation from openjdk-7.

2. CASE STUDY

Our case study sketches the capabilities of our data structure detection tool, `DSFinder`, and will help understand the detailed experimental results of Section 4. Our tool produces summary and detailed information about 1) probable implementations of data structures (both recursive data structures and arrays) and 2) uses of both system-defined and ad-hoc data structures. This section presents `DSFinder`'s output on a typical application, Apache `tomcat`, and briefly interprets it. Section 3 provides more technical details.

We have released `DSFinder` as free software and implemented a web interface to it. See

`http://www.patricklam.ca/dsfinder`

for more information on `DSFinder` and to try it out.

2.1 Data Structure Implementations

The main purpose of our tool is to count the number of data structure implementations in applications and libraries. In particular, it identifies all potential linked data structures by analyzing field structures. It then classifies these potential data structures using type and field name information.

Before describing our output on `tomcat`, we discuss the `LinkedList` class from Sun's openjdk-7 implementation of the Java Collections libraries. Figure 1 presents an excerpt from `LinkedList`. This implementation uses an inner class, `LinkedList$Entry`, which contains a recursive type definition—a potential data structure. (`HashMap` also uses an `Entry` inner class.) `DSFinder` finds that `LinkedList$Entry`'s fields `next` and `previous` belong to its whitelist and classifies `Entry` as a list. Our approach counts data structure implementations by counting the number of recursive type definitions (like `Entry`), and then counts data structure uses by counting the number of fields and instantiation sites of the containing classes (like `LinkedList`).

Figure 2 presents the summary results of our tool on version 6.0.18 of Apache Tomcat. In `tomcat`, our tool identifies 2 (likely) linked lists and 1 tree-like data structure, as well as 17 unidentified other potential (but unlikely) data structures, none of which are exact matches. (Figure 1 presented a typical exactly-matching linked list declaration, `Entry`.) The full results (not shown) indicate that the linked lists are `OrderInterceptor$MessageOrder` and `LinkObject`, while the tree-like data structure is `WebappClassLoader`. We found that, most of the time, "other" fields were not data structures, especially "other" fields that are not exact matches. Our tool includes these counts to be comprehensive. (Section 3.1 further explains our definition of data structures.)

Our classification of potential data structures into likely and unlikely data structures could be confounded by a number of phenomena, including different coding conventions and field names in non-English languages. Section 6 discusses these threats at length.

To identify the extent of ad-hoc data structures in the code, our tool also records the fact that 3 different classes (out of 656) contain data structure implementations. This implies that data structure

```
COUNTS OF IMPLEMENTATIONS
=========================

Linked lists                                            2
Parents/outers                                          1
Others (12 java.lang.Object, 5 non-Object fields)       17
 exact matching fields                                  0

Distinct classes containing linked lists and parents:   3

N-cycles                                                13

Arrays                                                  39
         read-only:                                     11
         w/arraycopy:                                   25
         hashtable-like:                                6
         (error bars:) [3]                              20

DECLARED SYSTEM COLLECTION FIELDS, BY IMPLEMENTING CLASS
========================================================
java.util.HashMap                                       62
java.util.ArrayList                                     20
java.util.Map                                           13
java.util.List                                          9
java.util.LinkedList                                    6
Others                                                  18


DECLARED AD-HOC COLLECTION FIELDS, BY IMPLEMENTING CLASS
========================================================
...apache.catalina.tribes.transport.bio.util.LinkObject 4
org.apache.catalina.loader.WebappClassLoader            3
...bes.group.interceptors.OrderInterceptor$MessageOrder 1
Others                                                  0


INSTANTIATED SYSTEM COLLECTIONS (counts of ‘new’ statements)
============================================================
java.util.ArrayList                                     230
java.util.HashMap                                       184
java.util.Hashtable                                     48
java.util.Vector                                        32
java.util.Properties                                    28
Others                                                  88


INSTANTIATED AD-HOC COLLECTIONS
===============================
...apache.catalina.tribes.transport.bio.util.LinkObject 2
...bes.group.interceptors.OrderInterceptor$MessageOrder 1
Others                                                  0


DECLARED COLLECTION PARAMETER TYPES [1]
=======================================
Collections are not data structures [2]                 23
Collections are potential data structures               72

 total org.apache.catalina.*                            8

java.lang.String                                        20
java.lang.Object                                        0

Ad-Hoc types:
=============
org.apache.catalina.Session                             2
org.apache.catalina.servlets.WebdavServlet$LockInfo     2
org.apache.catalina.realm.GenericPrincipal              1
org.apache.catalina.connector.Request                   1
org.apache.catalina.Executor                            1
Others                                                  1

System types:
==============
java.lang.String                                        20
java.util.ArrayList                                     1
java.lang.Runnable                                      1
java.util.Vector                                        1
java.util.Locale                                        1
Others                                                  1

TEMPLATE PARAMETERS                                     0
UNKNOWN                                                 128

[1] sums to more than count of non-array collections: consider HashMap<A,B>.
[2] e.g. class Foo { List<String> NotDataStructure; }
[3] number of counted arrays in classes with multiple arrays.
```

Figure 2: Summary results for `tomcat` benchmark.

manipulation is limited to a tiny part of the `tomcat` code.

Our tool also reports the number of "N-cycles", i.e. mutually recursive type declarations, in the program. An N-cycle occurs when class `C` contains a field of type `D`, and `D` contains a field of type `C`. In our experience, N-cycles do not form data structures.

Finally, our tool summarizes array usage in the application. Since arrays can be dynamically allocated, developers may use arrays to implement data structures (particularly hash tables). We counted the number of classes with array declarations in our benchmarks and collected some statistics on the use of these arrays. We found that many arrays are read-only: for instance, they might be passed into a class's constructor and stored as a field in that class. To help identify arrays that are actually used as data structures, we look for uses of `System.arraycopy`, which suggests list-like use of an array, as well as uses of the % operator and calls to `hashCode`, which suggest hashtable-like use of the array. We explain our array counts precisely, as well as the error bars data point, in Section 3. The `tomcat` application contains 39 fields with array type, of which approximately 11 are read-only arrays, 25 have calls to `arraycopy`, and 6 are hashtable-like. The error bars data point states that the reported counts may exceed actual counts by up to 20, due to approximations in our analysis.

2.2 Data Structure Uses

To better understand how programs use data structures, and in particular which data structures programs use in practice, our tool also collects two kinds of static counts about uses of data structures. It lists (1) the number of fields with collection types and (2) the number of `new` statements instantiating collections.

Field Counts.

To survey the use of persistent in-heap collections, we count the number of fields with declared collection types. We separate system collections (that is, subclasses of `java.util.Collection` or `java.util.Map`) from ad-hoc collections (e.g. classes which declare `next` fields). Note that our ad-hoc counts are approximate—they depend on the accuracy of our data structure implementation counts, as described above. We can see that in the `tomcat` benchmark, `HashMap` and `ArrayList` are the most commonly-used system collection types among fields, occurring respectively 62 and 20 times. Note that ad-hoc collections rarely appear as fields, which is consistent with their overall rarity in practice; the `LinkObject` type appears 4 times in field declarations, and there are only 4 other fields of ad-hoc collection type in the whole benchmark.

Instantiation Site Counts.

Counting instantiation sites rather than field declarations gives an alternate view of collection usage. Although we expect similarities between the results, note that instantiation sites will always use concrete types (e.g. `ArrayList`) while declarations can be of abstract types or interfaces (e.g. `List`). Our tool lists the most-frequently used instantiations of collection types. Again, we separate system collections from ad-hoc collections. In `tomcat`, the system collection `ArrayList` is instantiated at 230 different sites in the software, while `HashMap` is instantiated 184 times. (Note the inversion in order between `ArrayList` and `HashMap`.) Ad-hoc collections are instantiated 3 times. Interestingly, our tool does not detect instantiations of `WebappClassLoader`; searching through the code indicates that `WebappClassLoader` objects are created (using Java Reflection) from `newInstance` calls.

2.3 Composite Data Structures

Developers can create data structures such as graphs or trees using existing Java collections as building blocks. Our tool uses parametric type information, when it is available, to identify potential composite data structures. In `tomcat`, we can see that we have 72 potential composite data structures and 23 collections that we can rule out as being data structures (for instance, the 20 collections of `String`s are never data structures, barring some too-clever code on the part of the developer). We also print out the system and user-defined classes that occur most often as type parameters as well as the counts of formal template parameters appearing as parameter types. Observe that `Session` and `WebdavServlet$LockInfo` appear most often as user type parameters, while `String` appears most often as a system type parameter (by far). Finally, we print the number of unparametrized collections.

We manually inspected some benchmarks and found that most potential composite data structures are not data structures. For instance, in `tomcat`, we found that out of the 72 potential data structures, only 1 is an actual data structure. We include below an excerpt from the detailed `DSFinder` logs for that data structure, which happens to be a tree:

```
org.apache.catalina.core.ContainerBase:
  *java.util.HashMap* [protected] children
```

3. ANALYSIS

This section presents the static analyses behind our `DSFinder` data structure detection tool. It first describes the rationale and general methodology behind our approach, continues with a detailed presentation of the static analysis for linked heap data structures, and discusses our treatment of arrays. It also describes precisely how `DSFinder` counts uses of data structures in programs.

3.1 Rationale and Methodology

Our technique for automatically identifying heap data structures relies on the observation that data structures must be able to store arbitrarily-large collections of objects. Most data structures implement mutable sets. To be able to store arbitrarily-large sets, programs must use recursive type definitions or arrays. Because Java enforces type safety, we assume that a program's type definitions accurately reflect the program's use of memory.

We skip local variables and synthetic fields when counting data structure declarations. Java's local variables are unsuitable for defining data structures: local variable contents do not persist across method invocations, so it is hard to define recursive structures with local variables. Our tool only lists user-defined data structures in field declarations. We also skip synthetic fields as they are generated by the compiler, not defined in the source code, and thus cannot be used to implement data structures.

We briefly discuss "fixed-size data-structures". We are aware of two broad categories of such data structures: singletons providing data structure APIs, e.g. Java's `SingletonSet`, and fixed-universe data structures, e.g. bitvectors. We claim that singletons are not data structures, even if they present a data structure-like interface: their functionality differs significantly from mutable sets. Fixed-universe data structures typically must be initialized with the universe, and can then provide set operations based on this fixed universe. Fixed-universe implementations do fit our definition of a data structure, because the initialization phase can typically accept arbitrarily-sized sets; our tool would then detect the arrays created in the initialization phase.

`DSFinder` reads all class files belonging to specified Java packages and analyzes these class files using the ASM bytecode analysis framework [5]. We chose to consider all classes in specified packages (rather than all statically reachable classes) to better support dynamic class loading and reflection. Our results include all packages that an application's source files contribute to. Our package-based approach isolates the classes belonging to applications from libraries, which we counted as separate benchmarks.

3.2 Field-based Data Structures

We next present the key definition behind our data structure detection approach. Programs must use recursive type definitions or arrays (see Section 3.3) to store unbounded data in memory.

DEFINITION 1. *A class C contains a* recursive type definition *if it includes a field f with a type T, where T is type-compatible with C. Class C contains an* exact recursive type definition *if C contains a field f' with type C.*

Note that for `java.util.LinkedList`, the inner class `LinkedList$Entry` contains the recursive type definition, and we count the `Entry` type in our counts of implementations. However, when we later count uses, we instead search for uses of the containing `LinkedList` class.

While most data structures use exact recursive type definitions, developers may choose to implement data structures using non-exact recursive type definitions. Consider the following inner class of the `mxGraphModel` class from the `jgraph` benchmark:

```
public static class mxChildChange
             extends mxAtomicGraphModelChange {
  protected Object parent, previous, child;
  // ...
```

Clearly, the `mxChildChange` class implements a linked data structure; the containing class implements accessor methods which handle the heterogeneous types, casting as appropriate.

However, our experimental results indicate that such strange programming patterns are rare, and that developers usually use exact recursive type definitions to implement lists. Out of the 147 lists that we detected, only 28 of the lists used non-exact recursive type definitions, including 18 lists of `java.lang.Objects`. The data structures consisting of `Objects` came from 6 benchmarks: `bloat`, `hibernate`, `jchem`, `jgraph`, `sandmark` and `scala`. Non-exact type definitions (of superclasses, not `Object`) are more common when developers implement trees: it is useful to declare a `Node` class and populate the tree with `Node` subclasses.

Name-based Classification.

While type-based classification identifies all possible data structures[1], it errs on the side of completeness. Recursive type definitions are a coarse-grained tool for finding data structures in a large set of potential data structures. Understanding developer intent helps identify actual data structures.

Field names are a rich source of information about developer intent, and we implemented a simple set of heuristics that we found to be effective in practice. Our heuristics include both blacklists and whitelists: we (1) blacklist common false positives, (2) identify linked lists and trees, and (3) match a number of other field names that often denote data structures. `DSFinder` counts all remaining fields as unclassified potential other data structures.

We found that blacklists based on field names helped us to classify potential data structures. While (for completeness) `DSFinder` outputs all recursive type definitions, our experience with blacklists indicate that they work well in practice to classify data structures. Here are the blacklist criteria; fields which meet these criteria never form data structures in our benchmarks:

- Field type subclasses `Throwable`: Java programmers often subclass `Throwable` to declare custom exception types which re-throw existing exceptions.

- Field type is AWT or Swing-related: We observed many false positives with Swing and AWT. We found that this was due

[1]Developers could, of course, implement a virtual machine and code for that virtual machine, *à la* Emacs.

to Swing and AWT programming practices. For instance, many classes implement `JPanel` and themselves contain `JPanel` fields, but do not constitute data structures in the usual sense, because developers never used the `JPanel` hierarchy to store program data. We therefore chose to blacklist AWT and Swing field declarations. (SWT does not seem to lead to false positives.)

- Field type subclasses `Properties`: Java developers often create composite `Properties` classes which extend `java.util.Properties` yet themselves contain `Properties`. We found such implementations completely delegate property manipulations to the containee `Properties` objects, and therefore they do not constitute data structures.

- Field name contains `lock` or `key`: Such fields contain lock objects for synchronization.

- Field name contains `value`, `arg`, `data`, `dir`, `param`, or `target`: Such fields implement a one-to-one mapping between a container object and a containee. Usually the declared type is `Object` in such cases.

After the blacklist excludes fields, we run whitelists to explicitly include certain field names. These whitelists include entries for linked lists, trees, and graphs. Any recursive type declaration with field name containing `next` or `prev` counts as a linked list (only counting one list if both `next` and `prev` occur in the same class). Any recursive type declaration with a name containing `parent` or `outer` constitutes a tree-like data structure. (In our benchmarks, several containment hierarchies used fields named `outer` to implement containment.) We also list any fields named `child`, `edge` and `vertex` as probable data structures. Finally, we list any remaining fields which form recursive type declarations as "other" potential data structures, and inspect them manually.

Our whitelists are, in principle, subject to both false positives—where a developer names a field `next` but does not intend this field to form a data structure—and false negatives—fields that are not named `next` may also form lists. While false positives are possible with whitelists, we did not encounter any false positives in our data set (since the field must have both an appropriate type and name). False negatives do occasionally occur in our data set, since developers sometimes create linked lists using arbitrary field names. However, our tool outputs all recursive type definitions in its input, and our manual inspection identifies all false negatives.

One-to-many Relations.

Fields enable developers to implement one-to-one relations between objects. However, when implementing certain data structures such as trees or graphs, developers need to use one-to-many relations. Such relations call for the use of a data structure. To detect such (rare) composite data structures, our tool identifies cases where collections are known to contain elements of recursive type when parametric types are available. For instance, we list `jedit` class `InstallPanel$Entry`'s field `parents`, of type `List<InstallPanel$Entry>`.

Graph Data Structures.

Developers sometimes implement graph-like data structures using recursive type definitions and composite data structures. In the presence of parametric type parameters, `DSFinder` can identify potential composite data structures as graphs using type-based classfication, but it is unable to automatically classify them specifically as graphs, due to variations in graph implementations.

To estimate the frequency of graph implementations in programs without parametric type information, we performed a manual classification on 6 randomly-selected benchmarks out of the 53 benchmarks without template parameters (about 10% of the dataset). These benchmarks were `axion`, `bcel`, `xstream`[2], `log4j`, `jag` and `jfreechart`, altogether containing 2594 classes. Four of these benchmarks implemented 0 graphs, while `xstream` implemented 2 graphs and `axion` 3. We also used Eclipse's automatic type annotation tool to help us infer all parametric types in our `jedit` benchmark. The results in the part of `jedit` without parametric types were similar to the part with parametric types. We therefore believe that ignoring graphs on raw types does not significantly skew our counts of graph types.

We continued by examining `DSFinder` results for the benchmarks with parametric type information. We identified a number of graphs by looking over field names for common graph-related terminology. (When needed, we verified our conjectures by examining the code.) For example, `sandmark` declares this field:

```
public class CallGraphEdge
        implements sandmark.util.newgraph.Edge {
   private Object sourceNode, sinkNode; // etc.
```

The `CallGraphEdge` class clearly implements an edge in a graph connecting two vertices. However, `DSFinder` cannot automatically classify it as a graph: type-based classification can only classify `CallGraphEdge` as a potential data structure, because the fields are declared as `java.lang.Objects`. We believe that name-based classification for graphs would be unreliable, as the field names for graph nodes varied significantly in our dataset.

N-cycles.

Recursive type definitions capture immediate cycles in field declarations, e.g. cases where class `ListNode` contains a field `next` of type `ListNode`. Multiple classes may also participate in cycles, and Melton and Tempero have performed an empirical study of the incidence of cycles among classes [15]. Our notion of mutual recursion is similar to their notion of cycles among classes, but we are specifically looking for data structure implementations.

Figure 3 presents an instance of mutual recursion from the `artofillusion` benchmark. `Object3D` objects contain a reference to an object, `matMapping`, of type `MaterialMapping`, and `MaterialMapping` objects contain a reference to the `Object3D` which they are mapping. The `Object3D` and `MaterialMapping` types could be used together to implement linked structures in the heap, but the developers clearly intend to maintain the object invariant [12] that for an `Object3D` object `x`,

`x.matMapping.object = x.`

Note that this invariant, which states that the `object` field depends on the `matMapping` field, ensures that the mutual recursion does not constitute a data structure.

Our tool identifies many N-cycles in our benchmark set. However, we found that they generally did not implement data structures. Quite often, N-cycles relate a class and its inner classes, e.g. `Graph`, `Graph$NodeMap`, and `Graph$NodeList` in `bloat`. We believe that N-cycles generally do not implement data structures because it is individual classes (in conjunction with inner classes) that implement interfaces for Abstract Data Types [13].

3.3 Arrays

While some data structures, such as linked lists, are often implemented with recursive type definitions, other data structures, such as hashtables, are usually implemented using arrays.

[2]`xstream` contains one parametrized potential composite data structure which is not a data structure.

```
public abstract class Object3D {
  protected MaterialMapping matMapping;
  // ... other fields omitted ...

  public Object3D() { }
  public void setMaterial(Material mat, MaterialMapping map) { // ...
    matMapping = map; } }

public abstract class MaterialMapping {
  Object3D object; Material material;
  protected MaterialMapping(Object3D obj, Material mat) {
    object = obj; material = mat; }
```

Figure 3: Non-data structure mutual recursion among `artofillusion` classes.

```
private Item get (final Item key) {
  Item tab[] = table;
  int hashCode = key.hashCode;
  int index = (hashCode & 0x7FFFFFFF) % tab.length;
  for (Item i = tab[index]; i != null; i = i.next) {
    if (i.hashCode == hashCode && key.isEqualTo(i)) {
      return i; } }
  return null; }
```

Figure 4: Use of `hashCode` and modulo operator in `ASM` benchmark.

Lists can be implemented using either recursive type definitions (`LinkedList`) or arrays (`ArrayList`). To study how programs use arrays, we counted array-typed fields and classified these fields.

We decided to use the following two approximations for arrays: 1) we assume that code belonging to class `C` (and not subclasses) only accesses fields of class `C` (and not subclasses); and 2) we conflate accesses for all arrays declared in a single class. We verified approximation 1 by implementing an analysis which detected field accesses by objects of a different type (e.g. class `C` accesses `d.f`, where `d` is of type `D`)[3]. Such field accesses only arose in 14 benchmarks from our set of 62. Only in a few cases, such as the array-intensive benchmark `artofillusion`, did the number of instances exceed 50. We can therefore assert that the approximation is generally valid for our dataset. To understand approximation 2's effect, `DSFinder` counts how many arrays each class defines, and reports, as "error bars", the largest number of arrays in a class for each application. Because the error bars were relatively small for most applications (and nil for 26 of our benchmarks), we decided not to implement more precise counts of arrays without writes or `arraycopy` calls.

Preliminary investigations showed that many arrays were never modified inside a class; such arrays are initialized once and then never written to again. Immutable arrays are clearly not data structure implementations. Our tool therefore reports the number of arrays with no writes.

Two data structures that developers often implement using arrays are lists and hashtables. We use the following heuristics to detect these data structures: to find list implementations which are inserted to or removed from, we search for uses of `System.arraycopy`, and to find hashtables, we search for uses of modular arithmetic and calls to `hashCode()`. Figure 4 presents a hashtable from `ClassWriter` in the `ASM` library.

3.4 Data structure uses

To complement our work on implementations of data structures, we wanted to understand how often developers used data structures in their programs and which data structures they chose.

We survey two kinds of data structure uses: field declarations and object instantiations. We separate each of these uses into uses of Java collection types (implementers of the `java.util.Collection` and `java.util.Map` interfaces) and uses of instances of data structure implementations (as defined in Section 3.2). To count field declarations, we enumerate all fields of all classes in a benchmark and report fields with appropriate types. For object instantiations, we count occurrences of the appropriate `NEW` Java bytecode. Dynamic counts of instantiated objects are beyond the scope of this paper; Dufour et al. [7] investigated a number of dynamic metrics for Java benchmarks.

[3]Full results from this analysis are available on our webpage.

4. EXPERIMENTAL RESULTS

We next explore how programs used data structures in practice. We collected a suite of 62 open-source Java applications and libraries and applied our `DSFinder` tool (plus manual classification) to understand how these programs implemented and used data structures. In this section, we describe our experiments, and include remarks about our results. Next, in Section 5, we will propose four hypotheses and formally evaluate them based on our data.

Summary of Quantitative Results.

Table 1 summarizes our core findings. Our benchmark set includes Java programs from a wide variety of domains, including compiler compilers such as `javacc` and `sablecc`; integrated development environments such as `drjava`; databases such as (Apache) `derby`; and games such as `megamek`. Our benchmark set also includes the Apache `commons-collections` and the Java Runtime Environment itself (which contains 16 lists and 6 trees.) These benchmarks range in size from 3 classes (for `Bean`) to 5651 classes (for `azureus`).

For each benchmark in our benchmark set, we include counts of graphs, lists, and trees, as well as the total number of data structures (DS); we also include the number of classes in each program. Recall that the number of graphs is an underestimate, as discussed in Section 3. We also list the number of fields of data structure type ("declarations"), separated into system (SYS) data structures (which implement `Collection` or `Map`) and ad-hoc (AH) data structures (as previously identified by `DSFinder`), plus the number of classes containing arrays (ARR). To provide another perspective on data structure use, we also provide the number of instantiations of data structures and arrays, obtained by counting `new` and `newarray` bytecode instructions. Note that we add one to the "Instantiation AH" column for each instantiation of a class containing an ad-hoc data structure node (e.g. we count `LinkedList`s, not `LinkedList$Entry`s). Finally, we include more information on array usage in each of the benchmarks, including an estimate of the number of read-only arrays (RO), arrays which are used with `System.arraycopy` (w/AC), and arrays which occur along with calls to `hashCode` or `mod` operations (HS). (Note that ARR counts the number of classes with array declarations, while these counts estimate the number of arrays with the given properties.) The anticipated error for array measurements (ERR) is the largest number of arrays declared in any class in that benchmark.

To enable reproducibility of our results, we have included version numbers for each benchmark. Furthermore, we have also made the benchmark sources and all of our classifications available at the `DSFinder` website.

We used `DSFinder` along with manual classification to obtain the counts in Table 1. The manual classification took one author 3 days to perform. Note that no benchmark contains more than 16 list-like data structures, nor more than 24 linked data structures in all, and that the number of data structure definitions is tiny compared to the number of classes. Most of the ad-hoc data structures were lists. Programs declared fields of system data structures much more often than ad-hoc data structures. `jedit` breaks the trend, with extensive declarations of `org.gjt.sp.jedit.View` objects. These objects contain `prev` and `next` fields, so `DSFinder` automatically classifies `Views` as data structures. Benchmarks

Benchmark	Ver	Classes	Graphs	Lists	Trees	DS	Declarations		Instantiations				Arrays		
							SYS	AH	SYS	AH	ARR	RO	w/AC	HS	ERR
aglets	2.0.2	413	0	3	0	3	59	11	135	18	10	6	1	0	2
antlr-gunit (l)	3.1.3	147	0	0	0	0	2	0	20	0	0	0	0	0	0
aoi	2.7.2	680	0	11	5	16	26	81	247	209	103	9	20	36	33
argoUML	0.28	2068	0	2	9	11	87	34	1118	29	19	3	1	1	2
asm (l)	3.2	176	0	10	0	10	49	7	92	0	10	1	7	5	4
axion	r1.12	448	0	2	1	3	103	16	245	30	12	8	1	3	3
azureus	r1.97	5651	0	3	8	11	645	27	2077	15	169	53	41	41	47
bcel (l)	5.2	382	0	1	0	1	55	18	101	1	24	6	5	12	11
Bean	0.1	3	0	0	0	0	0	0	0	0	0	0	0	0	0
bloat (l)	1.0	332	0	7	16	23	128	23	361	4	22	2	7	5	4
cglib (l)	2.2	226	0	0	0	0	11	0	58	0	16	17	4	0	10
colt (l)	1.2.0	554	0	0	0	0	0	0	9	0	12	0	4	3	0
columba	1.4	1850	0	0	4	3	101	24	429	129	59	26	1	4	4
commons-cli (l)	1.2	23	0	0	0	0	12	0	17	0	1	1	0	1	0
commons-collections (l)	3.2.1	513	0	8	8	16	122	25	367	1	31	12	3	18	18
commons-lang (l)	2.4	127	0	1	0	1	14	1	62	1	7	3	1	2	0
commons-logging (l)	1.1.1	28	0	0	0	0	2	0	9	0	2	0	0	0	0
DCM	0.1	27	0	2	0	2	0	0	0	0	2	0	0	0	0
derby	10.5.1.1	1812	0	8	14	22	282	45	585	594	151	74	39	13	74
dom4j (l)	1.6.1	190	0	1	1	2	55	3	93	3	9	4	1	1	0
drjava	r4932	3155	0	5	15	19	107	31	951	70	50	36	0	15	17
fit	1.1	41	0	0	1	1	3	0	13	0	4	3	0	0	0
fop	0.95	1314	0	4	9	12	302	45	911	41	30	5	5	4	2
galleon	2.5.6	837	0	0	0	0	102	0	344	0	12	1	1	2	0
gantt	2.0.9	32	0	0	0	0	9	0	19	0	7	5	0	0	0
hibernate (l)	3.1.2	1143	0	2	4	6	344	13	643	37	92	82	14	6	62
hsqldb	1.8.0	242	0	4	3	7	5	25	11	61	26	8	10	15	17
ireport	3.0.0	2451	0	0	3	3	179	20	490	17	33	6	0	2	0
jag	6.1	344	0	0	2	2	45	0	191	0	11	7	0	0	2
jasper (l)	3.5.1	1437	0	33	0	33	381	58	612	16	75	35	9	2	23
javacc	4.2	155	0	5	2	7	31	47	206	42	9	4	2	1	0
jaxen (l)	1.1.1	213	0	0	4	4	17	2	53	10	0	0	0	0	0
jchem	1.0	914	0	3	0	3	212	9	567	13	43	3	11	17	8
jcm	r133	353	0	0	2	2	10	1	175	1	18	1	0	8	7
jedit	4.3pre16	1109	0	15	4	19	71	188	370	97	44	17	7	3	0
jeppers	20050608	78	0	0	0	0	0	0	0	0	2	0	0	0	0
jetty	6.1.17	331	0	5	3	8	97	25	187	39	29	8	2	18	12
jext	5.0	470	0	1	0	1	32	5	79	2	18	4	0	2	0
jfreechart (l)	1.0.13	819	0	0	3	3	144	8	326	2	34	7	12	14	19
jgraph (l)	0.99.0.7	177	0	4	4	8	28	6	108	10	7	4	1	1	2
jmeter	2.3.2	337	0	0	2	2	82	6	207	9	9	2	1	0	0
jre	1.5.0_18	712	0	16	9	25	92	116	284	251	33	10	10	9	6
jung (l)	2.0	487	12	0	0	12	131	0	404	1	3	4	0	0	3
junit	4.7	110	0	0	0	0	13	0	33	0	1	0	0	0	0
jython	2.2.1	953	0	4	3	7	66	12	284	154	71	47	18	25	36
LawOfDemeter	0.1	27	0	2	0	2	0	0	0	0	2	0	0	0	0
log4j (l)	1.2.15	259	0	2	1	3	40	9	67	0	15	4	1	1	0
lucene (l)	2.4.1	795	0	15	0	15	155	48	412	21	62	19	9	8	2
megamek	0.34.2	1799	0	0	0	0	23	0	978	0	82	11	8	15	8
NullCheck	0.1	139	0	2	0	2	0	0	0	0	2	0	0	0	0
pmd	4.2.5	720	0	6	5	11	21	43	269	34	17	7	1	0	2
poi (l)	3.2	1059	0	1	2	3	85	2	215	0	41	22	8	7	15
ProdLine	0.1	23	2	0	0	2	4	0	7	0	0	0	0	0	0
proxool (l)	0.9.1	105	0	1	0	1	24	3	52	1	5	0	1	0	0
regexp (l)	1.5	17	0	0	0	0	0	0	4	0	1	1	1	0	0
sablecc	3.2	285	0	0	1	1	129	1	792	0	13	6	0	3	0
sandmark	3.4.0	1087	4	10	22	36	272	27	996	18	86	49	10	20	34
StarJ-Pool	0.1	584	0	9	0	9	113	33	200	15	48	13	9	8	7
Tetris	0.1	31	0	0	0	0	1	0	1	0	1	0	0	0	0
tomcat	6.0.18	656	0	3	5	8	128	11	305	3	36	8	25	6	20
xerces	2.9.1	710	0	10	13	23	138	24	238	69	73	22	99	12	80
xstream (l)	1.3.1	342	0	0	2	2	71	4	154	1	8	2	3	1	2

Table 1: Array and data structure declaration and instantiation counts in benchmark set.

which we would expect to be numerically-intensive, such as `artofillusion`, a raytracer, and `jcm`, a Java climate model, indeed declare more arrays than collections. In terms of instantiations, our benchmarks always instantiated overwhelmingly more system collections than ad-hoc collections.

To summarize the array usage results, we see that many applications have lots of read-only arrays, up to three-quarters in `drjava`'s case, and measurements up to half are common. A non-trivial number of arrays appear to be used as data structures, but not a plurality. The error bars are reasonably low, giving some validity to our coarse analysis for understanding array usage.

Correlations between Size and Data Structure Count.

Figure 5 compares program size (measured in number of classes) against data structure count (from `DSFinder` plus manual classification) on our benchmark set. Note that we are counting both arrays and linked data structures in this figure: the reported number of data structures is the sum of the “data structures” column with the “hashtable-like” (HS) and “arraycopied” (w/AC) array columns.

Because the Pearson correlation coefficient is not robust to outliers, we chose to drop the `aoi`, `argoUML`, `azureus`, `derby`, `drjava`, `ireport`, and `xerces` outliers from the following calculations; for those benchmarks, either the number of classes or the number of data structures fell outside an outlier border of 1.5 times the inter-quartile range for that measurement. These data points greatly distorted the correlation coefficient and affect the slope of the line of best fit for applications by approximately 40%.

Our results indicate a medium linear correlation between a program's number of classes and its number of ad-hoc data structures; the Pearson correlation coefficient is 0.58. The slope of the line of best fit is 0.020 data structures per class.

We also computed separate lines of best fit for the sets of applications and libraries in our corpus. We found that our subcollections of libraries and applications, considered separately, each contain approximately 0.026 and 0.018 data structures per class respectively. However, the correlation coefficient for the libraries, at 0.74, is higher than the coefficient for the applications, at 0.52.

Collections Library Sufficiency.

We manually inspected our benchmark set to understand when developers implemented ad-hoc data structures instead of using system data structures. Developers generally implemented ad-hoc lists when they only needed limited functionality from the list structure; often, developers only add to and iterate over ad-hoc lists. Many ad-hoc lists implement hash table chaining. We conjecture that developers used ad-hoc lists for hash tables for perceived efficiency improvements. In our observations, ad-hoc list manipulations were confined to at most one class besides the defining class.

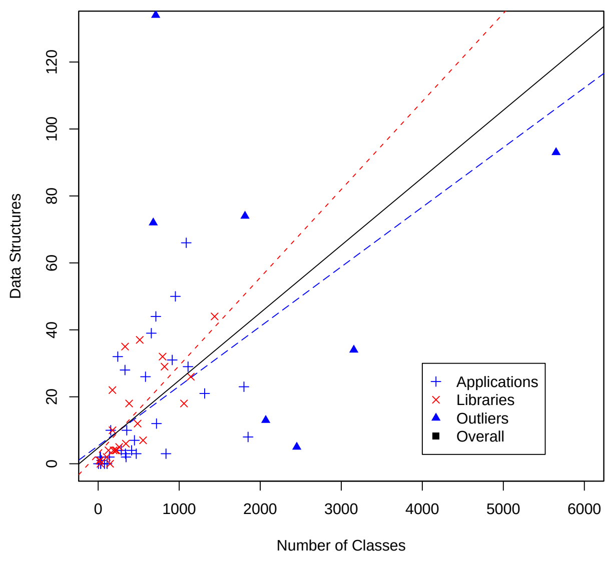

Figure 5: Benchmark size versus # of Data Structures

We also investigated the reason that developers implemented their own hash tables. Usually, these hash tables provide additional functionality that the system hash table does not provide; for instance, `jEdit` implements a hash table which can perform both case-sensitive and case-insensitive searches.

Based on their extensive usage, developers clearly use system data structures. Furthermore, developers often explicitly document their design decisions when implementing ad-hoc data structures. Developers most often documented ad-hoc data structures visible to external callers, and more rarely documented their decisions for private data structures.

Qualitative Observations.

We conclude this section by presenting some qualitative observations on our benchmark set. We describe two ad-hoc data structures which required manual classification and one declaration which does not constitute a data structure.

As mentioned previously, tree declarations are always ad-hoc. One tree occurs in `ArithExpr` from the `bloat` benchmark:

```
public class ArithExpr extends Expr {
  Expr left, right; // etc.
```

Note that the child nodes are of supertype `Expr`, which does not exactly match the `ArithExpr` type declaration. Other benchmarks also contain trees of expressions, notably `scala`, `sandmark` and `xerces`, and these trees also often do not contain exact type matches. We also found that libraries were more likely than applications to declare data structures based on interfaces (i.e. `Expr` might be an interface rather than a class.)

The `hsqldb` benchmark also contains a tree-like data structure of `Expression` nodes:

```
public class Expression {
    Expression eArg, eArg2; // etc.
```

This tree requires manual classification: while the fact that there are two `Expression` fields is suggestive of a doubly-linked list or tree, it is difficult to imagine a name-based whitelist entry which could understand the intent of this tree declaration.

Finally, we exhibit a recursive type declaration which does not constitute a data structure. This type of declaration occurs reasonably often, and can sometimes be blacklisted by name. Consider this example from the Apache commons primitives:

```
protected static class RandomAccessIntSubList
    extends RandomAccessIntList
    implements IntList {
  private RandomAccessIntList _list = null;
  // ...
```

In this case, the `_list` refers to a container object. However, by inspection, we can determine that the container object does not contain any further references to `RandomAccessIntList` or `RandomAccessIntSubList` classes, so that the heap references can form a chain of length at most 1. We therefore conclude that `_list` does not form a data structure. Sometimes such fields can form cycles, as seen in the earlier `Object3D/MaterialMapping` example, but those do not constitute data structures either.

5. DISCUSSION

Our experimental results allow us to test a number of hypotheses about data structure implementation and use. We next formulate four hypotheses and verify them against our experimental data.

HYPOTHESIS 1. *The number of data structure implementations varies linearly with the number of classes in a program.*

Recall from Section 4 that, after dropping outliers, the correlation coefficient between the number of classes and the number of data structures is 0.58. While the coefficient is far from 0, it neither strongly supports nor strongly rejects this hypothesis.

HYPOTHESIS 2. *Libraries implement more data structures than applications on a per-class basis.*

We were expecting to find far more data structure implementations among libraries than applications. The slope of the line of best fit was significantly higher for libraries, supporting this hypothesis.

HYPOTHESIS 3. *Developers use Java data structures more often than ad-hoc implementations.*

Our data certainly support this hypothesis. All but three of our benchmarks declare more fields of system data structure type than containers of ad-hoc data structure types, by at least a factor of two. The results from instantiations are even more overwhelming: all of the benchmarks instantiate more system data structures than ad-hoc data structures, at least when counting statically.

HYPOTHESIS 4. *Data structures are concentrated in a small portion of applications and libraries.*

Our data strongly support Hypothesis 4. The number of classes containing recursive type declarations for linked data structure implementations never exceeds 24, even for benchmarks with thousands of classes. Furthermore, data structures manipulations are well-encapsulated: Tempero [23] states that exposed fields are generally accessed from only one other class. This result is consistent with a commonly used model for manipulating data structures, in which a containing class (e.g. `java.util.LinkedList`) holds a reference to a "node" object (e.g. `java.util.LinkedList$Entry`). The "node" object contains the recursive type definition, while the containing class directly accesses and modifies elements of the node containee to perform standard data structure operations such as addition and removal of elements. If Tempero's conclusion applies to classes with data structure implementations, we could then conclude—based on our results—that data structure manipulations are confined to very small subsets of typical applications. We have also performed our own spot checks for data structure field reads, and our results are consistent with Tempero's results on our benchmark set.

6. THREATS TO VALIDITY

We survey threats to the validity of our hypotheses and discuss how we mitigate these threats. Some relevant threats include confounding factor and the composition of our corpus.

One possible threat to internal validity is that confounding factors, besides the independent variable, may affect dependent variables. Our independent variable is program complexity, as measured in number of classes, and our dependent variable is the number of data structures. Possible confounding factors include application domain; number and type of libraries used; and developer characteristics. While the confounding factors may indeed affect the observed number of classes, we believe that our large and varied benchmark set helps to control for these factors: we have chosen programs from many domains, and our applications have heterogeneous developer pools, so our results should not be skewed by these confounding factors.

Also, the number of classes may not be an ideal measure of program complexity; other software metrics might be better suited to measuring complexity and could better correlate with the number of data structures. However, we expect each class to implement at most one Abstract Data Type and to therefore use a small number of data structures. Note also that the number of classes is unambiguous, particularly easy to measure, and provides a rough estimate of the complexity of a software system.

Our system only analyses statically-available class files, not dynamically generated classes. We believe that dynamically generated classes would behave like the classes that we inspected.

One threat to construct validity is the accuracy of our manual classification. We believe that our manual classifications of data structures are fairly accurate; it is fairly straightforward to decide by inspection whether a field constitutes a data structure or not.

We next discuss potential barriers to generalizations of our analysis. Since our results are fairly conclusive on our corpus, the major threat to external validity is the representativeness of our corpus, which might not be representative of software projects in general.

Our corpus consists of over 60 open-source Java applications and libraries. The size of the corpus and the fact that it contains programs with over 5000 classes ensures that it represents Java programs of many different sizes. Even though our corpus only includes open-source programs, our results should also apply to proprietary applications. Melton and Tempero's empirical study [15] includes proprietary applications; these applications are generally similar to the open-source applications (although they do contain some outliers on some measurements).

Open-source applications might also be more likely to have unobfuscated English field names than applications in general. This threat applies only to `DSFinder`'s automatic classification of data structures. However, this threat should not affect our reported results, which are based on our manual classification of `DSFinder`'s exhaustive set of potential data structures.

We explicitly restricted our focus to Java applications. Different programming paradigms ought to yield different results. The important features of Java, for our purposes, are its object-oriented nature and its comprehensive class library. A study of C# applications should give similar results. On the other hand, C programs should contain far more data structure definitions: C is neither object-oriented nor endowed with collections in its standard libraries.

Scala [18] integrates the functional and object-oriented programming paradigms and produces Java Virtual Machine bytecode. Since our tool supports arbitrary Java bytecode, we examined version 2.7.4 of the Scala core library and compiler, which are themselves almost completely written in Scala. We found that Scala implemented 60 data structures over 7177 classes. Both the number and ratio of data structures to classes were far larger than anything in our input set, as we expected. Two possible reasons for this measurement are: (1) the Scala runtime system supports a separate language and therefore defines its own data structures, rather than using the Java data structures; and (2) in the functional programming paradigm, developers declare their own data structures more often than in the object-oriented paradigm.

7. RELATED WORK

We survey three classes of related work. We first describe some of the foundational work in abstract data types and encapsulation. Next, we discuss related work in the area of static and dynamic empirical studies of Java corpora. Finally, we explain how our work is applicable to research on sophisticated pointer and shape analyses.

7.1 Abstract Data Types & Encapsulation

Data abstraction, as first proposed by Liskov and Zilles in the context of operation clusters [13], is now a generally accepted program construction technique. Operation clusters are one ancestor of today's object-oriented programming languages. Data abstraction enables encapsulation of abstract data types into classes; Snyder [21] discusses some of the relationships between data abstraction, encapsulation, and design of object-oriented languages. Our work makes the assumption that data will be well-encapsulated, since it identifies and counts the classes which programs use to encapsulate data structures. In this study, we have manually examined some of our benchmark applications and found that data structure implementations are often (but not always) well-encapsulated[4]. Tempero's study of field use in Java [23] supports our belief: fields are rarely accessed by many outside classes.

Even if developers tend to respect encapsulation in practice, researchers have studied techniques for statically enforcing encapsulation. One example is work by Boyapati et al. [4] on the use of ownership types to enforce encapsulation. The problem is that some data structure fields cannot be declared "private": helper classes, such as iterators, or owner classes of "node" classes, legitimately need access to data structure fields. Ownership types enable designers to control such legitimate accesses. Work on ownership types would complement our research, since it would provide static guarantees that our surveys are exhaustive.

7.2 Empirical Studies

Researchers have recently conducted a number of studies of Java programs as-built. Collberg et al [6] presented quantitative information about the most-commonly-used classes and field types, as well as bytecode instruction profiles; our work overlaps theirs in that we also study most-commonly-used collection classes in applications, but our work also includes measurements of data structure implementations. Baxter et al [1] also present a quantitative study of software metrics for a large corpus of Java applications, but they instead attempt to determine whether these metrics fit a power-law distribution. Our research contributes to this growing body of empirical studies. We believe that our contribution will be particularly

[4]One flagrant counterexample occurs in the `hsqldb` benchmark, where five external classes access `org.hsqldb.Record` objects' `next` field, including three that insert records into the list.

useful for static analysis researchers, as it can help guide development of pointer and shape analysis techniques and tools.

Tempero [23] has investigated field visibilities and accesses on a substantial corpus of open-source Java programs. He finds that a surprisingly large proportion of applications include classes which expose instance fields to other classes, but few exposed fields are actually accessed by other classes. (We used this result earlier to support Hypothesis 4). Melton and Tempero also performed a related study [15] on cyclic dependencies between classes.

Malayeri and Aldrich [14] have empirically examined the uses of Java Collections in the context of structural subtyping to better understand which subset of Java Collections operations developers typically use. Our work studies a different aspect of Java Collections use: instead of trying to understand which parts of the Collections interfaces programs use, we study which Collections programs use, and which collections programs implement when they do not use Java Collections.

So far, we have discussed purely static, or compile-time, measurements of Java programs. Static measurements are a rich source of information about Java implementations and designs; in particular, they can identify the extent of data structure implementations and uses throughout Java programs. However, it is difficult to glean certain types of information about program behaviours from purely static counts. There is much work on profiling for Java. A particularly relevant example is the work of Dufour et al. [7], who present a set of dynamic metrics for Java programs. These metrics allow them to classify programs based on their behaviours. As in our work, some of their metrics concern arrays and data structures; however, their metrics—unlike ours—do not investigate the distribution of data structure manipulation code in programs, which is particularly valuable for software understanding and maintenance.

7.3 Sophisticated Pointer and Shape Analyses

Pointer and shape analyses have long been an active area of research, and many techniques and tools can successfully understand pointer manipulation code. The goal of shape analysis is to verify effects of sequences of heap-manipulating imperative statements; some examples include list insertion, concatenation, removal, sorting, and reversal code. The Pointer Analysis Logic Engine, PALE [16], and the Three-Valued-Logic Analyzer, TVLA [3] can successfully analyze properties of heap-manipulating code; for instance, these analyses can verify that the code maintains necessary invariants and always have the specified effects.

Due to their sophistication and precision, shape analysis algorithms are always computationally expensive, and traditional shape analyses do not scale up to entire programs. If one could limit the necessity for shape analysis to carefully-delimited fragments of software systems, then shape analysis would be a much more usable technique for verification tools and compilers. Our work contributes to this goal. Unfortunately, software does not yet come with modularity or encapsulation guarantees. Two approaches to enabling local reasoning are the use of Separation Logic [19] and modular verification, as seen for instance in the Hob and Jahob analysis frameworks [11, 24].

8. CONCLUSION

In this empirical study, we investigated the implementation and use of data structures in Java programs. We first defined recursive type definitions, which developers must use to implement ad-hoc data structures. We then presented our `DSFinder` tool, which identifies likely data structure implementations and prints out information about such implementations, field declarations of system and ad-hoc data structures, and instantiations of data structures, as well as array usage information. With our tool, we classified data structures in 62 open-source Java applications and libraries, and concluded that Java programs make extensive use of Java collections and limited use of ad-hoc data structures. No benchmark defined more than 24 ad-hoc data structures, and the number of data structures increased slowly in proportion to the number of classes.

We have established that most of the implementation of a Java program does not consist of manipulating arrays or linked data structures in the heap. A related question remains open: what makes up Java programs? What proportion of a Java program's implementation is simply boilerplate code, generated by an Integrated Development Environment?

9. REFERENCES

[1] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, M. Melton, and E. Tempero. Understanding the shape of Java software. In *Proceedings of the 21st OOPSLA*, pages 397–412, Portland, Oregon, October 2006.

[2] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 36–47, Atlanta, Georgia, November 2008.

[3] I. Bogudluv, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *Proceedings of Computer-Aided Verification*, pages 221–225, 2007.

[4] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, pages 213–223. ACM Press, 2003.

[5] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, November 2002.

[6] C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Software—Practice & Experience*, 37(6):581–641, May 2007.

[7] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of OOPSLA '03*, pages 149–168, Anaheim, California, October 2003.

[8] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proceedings of the 23rd POPL*, 1996.

[9] M. Henning. API design matters. *Communications of the ACM*, 52(5):46–56, May 2009.

[10] E. Kang and D. Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2008.

[11] V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *Transactions on Software Engineering*, 32(12):988–1005, December 2006.

[12] K. R. M. Leino and A. Wallenburg. Class-local object invariants. In P. Jalote and S. Rajamani, editors, *Proceedings of the 1st India Software Engineering Conference*, pages 57–66, February 2008.

[13] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM Symposium on Very High Level Languages*, pages 50–59, Santa Monica, California, 1974.

[14] D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. In G. Castagna, editor, *Proceedings of the European Symposium on Programming*, number 5502 in LNCS, pages 95–111, York, UK, March 2009.

[15] H. Melton and E. D. Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007.

[16] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

[17] R. O'Callahan and D. Jackson. Lackwit: a program understanding tool based on type inference. In *Proceedings of the 19th ICSE*, pages 338–348, 1997.

[18] M. Odersky. The Scala Language Specification, Version 2.7. www.scala-lang.org/docu/files/ScalaReference.pdf, 2009.

[19] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–71, Copenhagen, Denmark, July 2002.

[20] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[21] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In N. Meyrowitz, editor, *Proc. 1st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 1986*, pages 38–45, Portland, Oregon, October 1986.

[22] Sun Microsystems Inc. Collections framework overview. http://java.sun.com/j2se/1.4.2/docs/guide/collections/overview.html, 1999. Last accessed on 16 July 2009.

[23] E. Tempero. How fields are used in Java: An empirical study. In *Proceedings of the Australian Software Engineering Conference*, pages 91–100, Gold Coast, Australia, April 2009.

[24] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *Proc. PLDI*, 2008.