Implementation, Use, and Sharing 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. The well-organized use of standard data structure implementations contributes to good modularity. We empirically investigate this aspect of modularity—namely, the implementation, use, and sharing 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 and interfaces 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.

XXX something about exposure/sharing.

1. INTRODUCTION

Data structures are central to many software systems. Classically, programmers implement in-memory data structures with pointers. Understanding a software system typically requires understanding how the system manipulates data structures as well as understanding the relationships between its different data structures. Well-designed software systems with modular, well-encapsulated data structures are clearly easier to understand and maintain than systems where data structures are shared between and manipulated by many disparate parts of the code. Also, no matter how well-encapsulated the data structure manipulations may be, such code always poses a challenge to static analysis techniques, as it requires intricate reasoning about the code's behaviour.

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 [23] 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 goals of our research are: 1) to understand how often developers implement ad-hoc data structures versus library data structures, and 2) to estimate how widely programs share data structures between their different modules. 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? Beyond the implementation style, we also seek to understand whether data structure manipulations are confined to a small set of classes, or whether data structures are accessed and updated by dozens of different classes?

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 modularity as well as how programs are built.

In this paper, we present the results of our analysis of data structure implementation, interfaces, and exposure in a corpus of 62 open-source Java programs and libraries. Our work was driven by the following three hypotheses.

Hypothesis 1: Implementations. It is possible to automatically identify data structure implementations. Pointer-based data structure implementations are extremely rare.

Hypothesis 2: Interfaces. It is possible to automatically identify data structure interfaces. Interfaces are ubiquitous.

Hypothesis 3: Sharing. While many collections are exposed to clients, the actual sharing of changing collections between program modules is rare.

To explore these hypotheses, we developed a number of definitions for data structure implementations and interfaces. We have developed an analysis tool which identifies data structure implementations, interfaces, and exposure. To identify implementations, it searches 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 performs a simple dataflow analysis to identify classes which delegate to underlying data structure implementations. We found that a generalization of this analysis gave us some insight into the potential sharing of data structures between different modules of programs.

Sharing is important: to understand a program, we need to understand its data structure uses, and particularly the distribution of these uses across different modules. A data structure which is updated in many unrelated areas of a program would make that program much less modular.

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, 9, 15, 21]) 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.

  Some form of assume-guarantee reasoning, as embodied for instance in the Hob and Jahob systems [11, 27], or the use of separation logic [20], is key to the successful deployment of heavyweight static analysis 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 [18] 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.

Our paper makes the following contributions:

- We propose the concepts of identifying possible data structure implementations by type declarations and classifying probable data structures using field and type information, as well as identifying data structure interfaces by noticing that they delegate to data structure implementations.

- 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, use, and share data structures.

2. MOTIVATION

We motivate this paper by presenting a sequence of data structure implementation and usage styles. We discuss the effects of each of these styles on modularity, program understanding, and analyzability. Later in the paper, we will describe our empirical findings about the prevalence of each of these styles in our benchmark suite.

Style 1: C-style linked lists.

The least modular data structure implementation style is the C-style linked list, as shown in Figure 1.

```
typedef struct _FcValueList *FcValueListPtr;

typedef struct _FcValueList {
    struct _FcValueList *next;
    FcValue              value;
    FcValueBinding       binding;
} FcValueList;
```

Figure 1: C-style linked list implementation from `fontconfig`.

Any procedure which gets a `FcValueListPtr` may navigate or modify the linked list by manipulating the `next` pointer values. Such lists are difficult to reason about because list manipulation code could well crosscut the main codebase. (In Java, even if the `next` field were private, the class could still expose a `setNext()` method.)

Furthermore, there are many valid ways to express linked list manipulation, which complicates both code understanding and the development of static analysis tools—analyzing linked list manipulations requires shape analysis techniques. Finally, because each linked list node can serve as the head of a linked list, it can be difficult to distinguish the identities of different lists in the program.

One of the goals of our research is to empirically demonstrate that C-style linked lists are rare in Java programs, due to the availability of Java collections and arrays, and to understand the circumstances under which developers do use ad-hoc linked lists. Reasoning about calls to encapsulated methods like `List.add()` is much easier for both developers and static analysis tools than reasoning about pointer manipulations. In particular, our primary goal in this research is to demonstrate that data structure interfaces suffice for reasoning about the contents of the vast majority of data structures in Java programs.

Note that we know how to modularize these cross-cutting concerns, using the ancient techniques of procedural abstraction and object-oriented programming. This research investigates whether developers actually use these mechanisms, or, alternatively, the reasons behind their decisions to expose the representations.

Style 2: Shared collections.

We expect the "shared collections" style to be dominant; part of our work attempts to explore the modularity of Java programs by understanding the extent of collection sharing between different program modules. This style includes Java collections that occur as potentially-exposed fields of classes.

```
public class GoBackConfiguration {
  private List<NameValue> mPaths =
              new ArrayList<NameValue>();
  private boolean mModified;

  public List<NameValue> getPaths() { return mPaths; }

  public void setPaths(List<NameValue> value) {
    mModified = true; mPaths = value; }

  public void addPath(NameValue nameValue) {
    mModified = true; mPaths.add(nameValue); }
  /* etc... */ }
```

Figure 2: Shared `mPaths` collection from `galleon`.

Figure 2 presents an example of a shared collection. The `mPaths` collection is shared because the contents of the collection could, in theory, be modified outside the defining `GoBackConfiguration` class: the `getPaths()` method returns a reference to the list itself, which clients can modify. Also, a client can change `mPaths` by calling the `setPaths()` method. Even though the field is private, the public `getPaths()` and `setPaths()` methods imply that the field is not fully-encapsulated: reasoning about the contents of `mPaths` requires reasoning about callers to `getPaths()`, `setPaths()` and `addPath()`.

This code appears to maintain the invariant that `mModified` must be set to true after any change in the state of the `GoBackConfiguration`. Note that the exposure of the `mPaths` list in `getPaths()` allows clients to potentially violate this invariant; to verify the invariant, an analysis must inspect each caller to `getPaths()` and ensure that it does not modify the list. (In this case, callers to `getPaths()` do not, in fact, modify the list.) Alternatively, the developer of the `GoBackConfiguration` class could modify the code by wrapping a call to `Collections.unmodifiableList()` around the return value to `getPaths()`. Such defensive programming would fully encapsulate the `mPaths` collection.

Style 3: Fully-encapsulated collections.

The `addPath()` method from Figure 2 partially encapsulates the `mPaths` collection by allowing clients to modify the collection without getting a reference to it. However, that collection is potentially shared between (and modified in) different classes. Another programming style would prevent all external accesses to collections and only allow access to collection elements through a well-defined API. Data structure implementations are typically fully-encapsulated: they only allow clients access to the internal representation of the data structure through their public interfaces. Other classes then use the data structure implementations as building blocks, delegating manipulations to the implementation methods. Although we expect that the shared-collection style will be dominant, our research measures how many collections are shared and how many are fully-encapsulated in Java programs.

Figure 3 presents an example of a fully-encapsulated collection in the Soot compiler framework. Even though a client may get a reference to the `parameterTypes` field through the `getParameterTypes()` method, the reference is to an unmodifiable copy of the original list. Note that the contents of any list returned from `getParameterTypes()` will never change. Furthermore, all writes to the list, which must go through `setParameterTypes()`, create a new unmodifiable copy of the list and store that copy to the field. Even the setter of the parameter types cannot modify the list it has stored.

Analyzing fully-encapsulated collections is relatively straightforward. To find all places where the contents of a collection with suitable visibility may change, it suffices to inspect the containing class for all methods which may change the contents of the collection; a simple call-graph traversal can then identify the callers of such methods.

```
public class SootMethod {
  List parameterTypes;

  public SootMethod(..., List parameterTypes, ...) {
    this.parameterTypes = new ArrayList();
    this.parameterTypes.addAll(parameterTypes);
    this.parameterTypes =
      Collections.unmodifiableList(this.parameterTypes); }

  public int getParameterCount()
    { return parameterTypes.size(); }

  public Type getParameterType(int n)
    { return (Type) parameterTypes.get(n); }

  public List getParameterTypes()
    { return parameterTypes; }

  public void setParameterTypes( List l ) { // excerpt...
    List al = new ArrayList(); al.addAll(l);
    this.parameterTypes = Collections.unmodifiableList(al);
    subsignature = ...;  } }
```

Figure 3: Fully-encapsulated `parameterTypes` collection from `soot`.

Style 4: Immutable functional-style collections.

In the purely-functional programming style, collections are created once and never mutated; computation proceeds by creating modified copies of existing collections. The Polyglot compiler framework [17] is an example of a Java application largely programmed in a purely-functional style. We expect programs in this style to be rare.

Figure 4 presents an example of an immutable data structure from `lucene`. The constructor populates the `fieldSelections` field from its arguments, while the `accept()` method queries the contents of the map based on its argument.

We do not expect to find widespread use of functional-style collections. Immutable collections resemble fully-encapsulated collections, except that they lack methods which change either the reference or the contents of the underlying collections. Analyzing immutable collections would be about as difficult as analyzing fully-encapsulated collections, except that an analysis of an immutable collection would not have to deal with changes to the collection.

2.1 Implications

We discuss the implications of this work at two levels. First, the DSFinder tool aids in classifying and understanding individual programs. More broadly, this research enables a collection interface-based approach to static analysis.

At the specific level, DSFinder identifies all potential recursive data structures in a program and characterizes how a program manipulates its data structures. This information can help maintainers familiarize themselves with new programs and orient themselves with the dominant style of programming in a program. For in-

```
public class MapFieldSelector implements FieldSelector {
  Map fieldSelections;

  public MapFieldSelector(Map fieldSelections)
    { this.fieldSelections = fieldSelections; }

  public MapFieldSelector(List fields) {
    fieldSelections = new HashMap(fields.size()*5/3);
    for (int i=0; i<fields.size(); i++)
      fieldSelections.put(fields.get(i),
                          FieldSelectorResult.LOAD); }

  public FieldSelectorResult accept(String field) {
    FieldSelectorResult selection =
        (FieldSelectorResult) fieldSelections.get(field);
    return selection!=null ?
        selection : FieldSelectorResult.NO_LOAD; } }
```

Figure 4: Immutable `HashMap` from `lucene`.

stance, if the program takes care to always fully-encapsulate its data structures, then a maintainer should continue to ensure that new data structures are also fully-encapsulated; failure to do so will lead to maintenance headaches later on.

This work also has implications for the research community. A number of recent papers rely on collection interfaces, including thin slicing [8], object histories [16], and other analyses of data structures [25, 26]. These papers make the implicit assumption that it is easy to identify data structure manipulations. The present work shows that this assumption is indeed valid: static analyses do not have to resort to shape analysis techniques to identify a myriad of data structure manipulations scattered through the code. We instead show that it is, for the most part, appropriate to reason about program behaviour using data structure interfaces, and we propose an algorithm for automatically identifying such interfaces in Java programs, as well as exceptions to this general rule.

3. DEFINITIONS

The sequence of examples in Section 2 motivate the following definitions of data structure interfaces and implementations. Section 4 presents a series of static analyses which use these definitions to 1) identify data structure implementations and interfaces and 2) evaluate the extent to which data structures are shared.

Data structure implementations.

We next present the key definition behind our data structure detection approach. Programs must use recursive type definitions or arrays 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.*

DEFINITION 2. *A class C* implements a data structure *if it contains a recursive type definition or an* `Object`*-typed array.*

Figure 5 presents an excerpt from the `openjdk` definition of the library `LinkedList` class. This implementation uses an inner class, `LinkedList$Entry`, which contains a recursive type definition—a potential data structure. Our approach counts data structure implementations by counting the number of recursive type definitions (like `Entry`).

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 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 5: `LinkedList` implementation from openjdk-7.

```
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.

Our tool counts 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, as described in Section 4.

Data structure interfaces.

Data structure interfaces provide modular ways for clients to access or manipulate the contents of data structures. We next define the notion of a data structure interface.

DEFINITION 3. *A method m is a* data structure interface method *if at least one of the following hold:*

1. *m's containing class implements a Java Collection interface and m implements a method from that Collection interface;*
2. *m is a data structure implementation method; or,*
3. *m delegates to a data structure implementation or interface method m': that is, m calls m', passing m' one of its parameters, or m returns the return value of m'.*

A class C provides a data structure interface *when it contains one or more data structure interface methods. If C contains N distinct Collections and delegates to each of them, it provides N data structure interfaces. Furthermore, C* provides an immutable data structure interface *if it does not provide any methods which modify the underlying data structure.*

We also propose the following definition of a data structure implementation method.

DEFINITION 4. *A method m is a data structure implementation method if it accesses a non-primitive type array (e.g.* `Object[]`*) or if it reads a recursive-type field.*

To summarize, our definition identifies methods that 1) claim to perform data structure manipulations (they implement the appropriate interface) or 2) actually carry out or delegate to methods which perform data structure manipulations. Section 4 describes the static analysis that we implemented to identify data structure interface methods.

Modularity and exposure.

Our third research hypothesis examines the modularity of data structures; the following definition enables us to distinguish between the shared and fully-encapsulated collections of Section 2.

DEFINITION 5. *A collection is* exposed *if it is possible to read or write the reference to the collection from outside the class.*

A field `f` exposes the collection that it contains if it is accessed outside its defining class, or if some method inside the class returns the value of `f`.

4. 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 continues with our analyses to detect data structure interfaces and data structure exposure.

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.

4.1 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 which are generated by the compiler, not defined in the source code, and thus cannot be used to implement data structures.

`DSFinder` reads all class files belonging to specified Java packages and analyzes these class files using the Soot Java 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.

4.2 Finding Data Structure Implementations

We use a simple static analysis based on definition 1 to detect C-style linked lists (style 1). However, while the definition captures all possible recursive data structures, we found that some heuristics reduced the false-positive rate and helped us focus on actual data structures. The three heuristics we used were: 1) prioritizing exact recursive type definitions; 2) rejecting (blacklisting) potential data structures by type or field name; and 3) accepting (whitelisting) common field names.

Exact recursive type definitions.

Our experimental results indicate 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. Our tool therefore (1) blacklists common false positives, (2) identifies linked lists and trees, and (3) matches 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 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. (The Eclipse SWT library 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

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

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.

Graph Data Structures.

Developers sometimes implement graph-like data structures using recursive type definitions and already-existing data structures or arrays. In the presence of parametric type parameters, `DSFinder` can identify potential composite data structures as graphs using type-based classification, 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`, `log4j`, `jag` and `jfreechart`, altogether containing 2594 classes. Four of these benchmarks implemented 0 graphs, while `xstream` implemented 2 graphs and `axion` 3.

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.

Anecdotes.

We discuss three examples from our benchmark suite. 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.

4.3 Finding Data Structure Interfaces

The next phase of our research sought to identify data structure interfaces. Data structure interfaces underlie styles 2, 3 and 4 from Section 2. Furthermore, the ability to automatically identify interfaces enables analyses to reason about these interfaces without also understanding the underlying implementation; the analysis of the implementation could be outsourced to a separate analysis, or the implementation could simply be assumed correct (an uncontroversial assumption for Java Collections).

Definition 3 proposed a notion of a data structure interface method. We now briefly explain the static analysis that we used to identify data structure interfaces. Clearly, no static analysis is required to identify a method m which implements a Java Collection interface. The two fundamental interfaces are `java.util.Collection` and `java.util.Map`; all other Java Collections declare at least one of these interfaces. Also, a simple inspection of the statements in method m reveals whether any of these statements access an array or a recursive-type field—we keep a list of these fields from the analysis in Section 4.2.

To determine whether a method delegates to a data structure implementation method, we perform a simple forward dataflow analysis which intraprocedurally tracks the origin of each local variable in a method. Two common origins for variables are method parameters and return values of called methods. (Others include fields and caught exception values). Then, we say that a method m delegates to a data structure implementation if it contains a method invocation which calls a data structure implementation method m' and passes m' one of its method parameters. Alternatively, m also delegates if it returns a value obtained as a return value from a data structure implementation method.

To count the number of distinct collections in a data structure interface, we count fields with types that implement `java.util.Collection` and `java.util.Map`. Each delegation to a distinct collection accounts for a separate data structure interface.

4.4 Modularity, Exposure and Immutability

We use a similar static analysis as the one for delegation to determine whether classes expose data structures or not. In this analysis, we check whether collection fields are returned from methods or not. (Although this is not the only way to expose a field, we believe that it provides a reasonable estimate of field exposures.)

Shared, encapsulated and immutable collections.

We now have the machinery to differentiate between styles 2, 3 and 4 from Section 2. Shared collections are data structure interfaces which expose the underlying interface. Fully-encapsulated collections do not expose the underlying interface. We do not

currently count the number of immutable collections, although it would be a straightforward extension of our current technique—it would suffice to count collections which provide no access to write methods.

5. EXPERIMENTAL RESULTS

We next explore how programs implement, use, and share 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, used, and shared data structures. In this section, we describe our experiments and include remarks about our results.

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 4. 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). 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.)

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.

XXX new numbers

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 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.

5.1 Data Structure Implementations

We first used our `DSFinder` tool to identify C-style linked lists, and manually verified the results of `DSFinder`'s output in ambiguous cases. Recall that manipulations of these lists require shape analysis techniques to statically analyze. The first set of numbers in Table 1 presents our counts of data structure implementations.

We found that no benchmark in our suite contained more than 16 linked lists, and that benchmark was the Java library itself, which contains the data structure implementations that most programs themselves use. We also see that 26 programs contained no programmer-defined linked lists at all, including the substantial `megamek` and `columba` applications, which each contain about 1800 classes. The remaining programs contain between 1 and 15 list definitions.

To better understand the circumstances under which programmers defined ad-hoc C-style linked lists, we manually investigated a number of programs which contained lists. We continue by describing the results of our exploration on two selected benchmarks, `axion` and `lucene`.

Benchmark 1: `axion`.

This benchmark is a small relational database system implemented in Java. We found that it contains 2 list-containing classes and 1 tree-containing class. The list-containing classes are `IntHashMap$Entry` and `Token`: these developers seem to have implemented a hash map based on primitive `int` types before the advent of Java 1.5 autoboxing; and the automatically generated JavaCC code contains a `next` link between tokens. This application also contains a tree of `Objects` in a class called `FromNode`.

The `Entry` class contains a `_next` field so that it can chain elements. This field is private, but `Entry` allows public read-only access to its `_next` and `_prev` fields via getter methods. The `IntHashMap` returns `Entry` objects upon request, but no callers request such objects. The list of `Entry` objects is therefore fully-encapsulated in the sense that external classes may not modify the list. (One of the most common use cases for implementing lists across our benchmark set was in implementing custom hash tables.)

Benchmark 2: `lucene`.

We also explored data structure implementations in one of the two benchmarks with the most list implementations, `lucene`, a text search engine library. This benchmark contains 12 lists, 2 of which occur in JavaCC-generated `Token` code. An additional 4 lists occur in chaining implemenations of hash tables; the three hash tables `TermsHash`, `TermsHashPerThread` and `TermsHashPerField` are closely related, while the `Bucket` inner class of `BooleanScorer` is separate.

The deprecated `HitDoc` class implements a doubly-linked list of cached documents. The `HitDocs` themselves are encapsulated behind an `Iterator` which returns hits on demand. `HitDocs` have been replaced by `TopDocCollector`, which instead uses a standard array-based `PriorityQueue` object to store top documents. The methods in `PriorityQueue` fit our definition of data structure implementation methods.

Another example of a list in `lucene` occurs in the `SpansCell` inner class, which contains a private `next` field to form linked lists. It happens that the containing class also contains a `PriorityQueue` which also stores a list of spans, but these data structures are sorted by different criteria. The encapsulated C-style linked list in `SpansCell` definitely requires significant programmer effort to understand (particularly in its relationship to the

Benchmark	Ver	Classes	Graphs	Lists	Trees	DS	Declarations		Interfaces	Exposed DS	Arrays			
							SYS	AH			ARR	RO	w/AC	HS
aglets	2.0.2	413	0	3	0	3	59	11	21	3	10	6	1	0
antlr-gunit (l)	3.1.3	147	0	0	0	0	2	0	0	0	0	0	0	0
aoi	2.7.2	680	0	11	5	16	26	81	0	0	103	9	20	36
argoUML	0.28	2068	0	2	9	11	87	34	0	0	19	3	1	1
asm (l)	3.2	176	0	10	0	10	49	7	24	5	10	1	7	5
axion	r1.12	448	0	2	1	3	103	16	0	0	12	8	1	3
azureus	r1.97	5651	0	3	8	11	645	27	0	0	169	53	41	41
bcel (l)	5.2	382	0	1	0	1	55	18	14	3	24	6	5	12
Bean	0.1	3	0	0	0	0	0	0	0	0	0	0	0	0
bloat (l)	1.0	332	0	7	16	23	128	23	0	0	22	2	7	5
cglib (l)	2.2	226	0	0	0	0	11	0	0	0	16	17	4	0
colt (l)	1.2.0	554	0	0	0	0	0	0	9	0	12	0	4	3
columba	1.4	1850	0	0	4	3	101	24	23	1	59	26	1	4
commons-cli (l)	1.2	23	0	0	0	0	12	0	0	0	1	1	0	1
commons-collections (l)	3.2.1	513	0	8	8	16	122	25	0	0	31	12	3	18
commons-lang (l)	2.4	127	0	1	0	1	14	1	0	0	7	3	1	2
commons-logging (l)	1.1.1	28	0	0	0	0	2	0	0	0	2	0	0	0
DCM	0.1	27	0	1	1	2	0	0	0	0	2	0	0	0
derby	10.5.1.1	1812	0	8	14	22	282	45	5	0	151	74	39	13
dom4j (l)	1.6.1	190	0	1	1	2	55	3	0	0	9	4	1	1
drjava	r4932	3155	0	5	15	19	107	31	0	0	50	36	0	15
fit	1.1	41	0	0	1	1	3	0	2	1	4	3	0	0
fop	0.95	1314	0	4	9	12	302	45	0	0	30	5	5	4
galleon	2.5.6	837	0	0	0	0	102	0	0	0	12	1	1	2
gantt	2.0.9	32	0	0	0	0	9	0	1	1	7	5	0	0
hibernate (l)	3.1.2	1143	0	2	4	6	344	13	0	0	92	82	14	6
hsqldb	1.8.0	242	0	4	3	7	5	25	0	0	26	8	10	15
ireport	3.0.0	2451	0	0	3	3	179	20	0	0	33	6	0	2
jag	6.1	344	0	0	2	2	45	0	1	1	11	7	0	0
jasper (l)	3.5.1	1437	0	0	33	33	381	58	0	0	75	35	9	2
javacc	4.2	155	0	5	2	7	31	47	7	2	9	4	2	1
jaxen (l)	1.1.1	213	0	0	4	4	17	2	0	0	0	0	0	0
jchem	1.0	914	0	3	0	3	212	9	3	0	43	3	11	17
jcm	r133	353	0	0	2	2	10	1	0	0	18	1	0	8
jedit	4.3pre16	1109	0	15	4	19	71	188	180	16	44	17	7	3
jeppers	20050608	78	0	0	0	0	0	0	0	0	2	0	0	0
jetty	6.1.17	331	0	5	3	8	97	25	11	2	29	8	2	18
jext	5.0	470	0	1	0	1	32	5	2	0	18	4	0	2
jfreechart (l)	1.0.13	819	0	0	3	3	144	8	26	1	34	7	12	14
jgraph (l)	0.99.0.7	177	0	4	4	8	28	6	0	0	7	4	1	1
jmeter	2.3.2	337	0	0	2	2	82	6	0	0	9	2	1	0
jre	1.5.0_18	712	0	16	9	25	92	116	124	15	33	10	10	9
jung (l)	2.0	487	12	0	0	12	131	0	0	0	3	4	0	0
junit	4.7	110	0	0	0	0	13	0	0	0	1	0	0	0
jython	2.2.1	953	0	4	3	7	66	12	112	6	71	47	18	25
LawOfDemeter	0.1	27	0	2	0	2	0	0	0	0	2	0	0	0
log4j (l)	1.2.15	259	0	2	1	3	40	9	3	1	15	4	1	1
lucene (l)	2.4.1	795	0	13	0	13	155	48	26	4	62	19	9	8
megamek	0.34.2	1799	0	0	0	0	23	0	0	0	82	11	8	15
NullCheck	0.1	139	0	2	0	2	0	0	0	0	2	0	0	0
pmd	4.2.5	720	0	6	5	11	21	43	18	5	17	7	1	0
poi (l)	3.2	1059	0	1	2	3	85	2	0	0	41	22	8	7
ProdLine	0.1	23	2	0	0	2	4	0	7	1	0	0	0	0
proxool (l)	0.9.1	105	0	1	0	1	24	3	0	0	5	0	1	0
regexp (l)	1.5	17	0	0	0	0	0	0	0	0	1	1	1	0
sablecc	3.2	285	0	0	1	1	129	1	47	0	13	6	0	3
sandmark	3.4.0	1087	4	10	22	36	272	27	6	1	86	49	10	20
StarJ-Pool	0.1	584	0	9	0	9	113	33	5	0	48	13	9	8
Tetris	0.1	31	0	0	0	0	1	0	0	0	1	0	0	0
tomcat	6.0.18	656	0	3	5	8	128	11	2	0	36	8	25	6
xerces	2.9.1	710	0	10	13	23	138	24	3	0	73	22	99	12
xstream (l)	1.3.1	342	0	0	2	2	71	4	0	0	8	2	3	1

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

queue).

In general, we found that `lucene` contained quite a few data structures. The difficulty may originate in the fact that `lucene` attempts to perform algorithmically complicated tasks. Overall, this benchmark seems to be quite challenging to understand and analyze, both from a maintenance point of view as well as from a static analysis point of view. Nevertheless, only 12 classes contain field declarations consistent with lists, a small fraction of the 795 classes in the entire benchmark. (A couple more classes manipulate these lists). Fortunately, almost all of our other benchmarks were less complicated than `lucene`.

Benchmark 3: `aglets`.

We also investigated the `aglets` benchmark, a mobile agent platform, for its use of data structures. This benchmark contains 413 classes in all. Among these classes, it contains three C-style list declarations, in the `DeactivationInfo`, `MessageImpl` and `CacheManager` classes. We examined these declarations in more detail.

The `MessageImpl` class defines a linked list of `MessageImpl` message objects which are stored in a `MessageQueue`, which contains the list manipulation code. Comments in the `MessageQueue` class also indicate that the class should be replaced by a priority queue and label the `insert()` method with "problem!". The internal list is also accessed in the `MessgeManagerImpl` class, also in the same package.

The `DeactivationInfo` class serves as an entry in a list of agents to be woken up in the future. Its `next` field is read by one other class in the same package.

Finally, the `CacheManager` class contains a private `LinkedList`, which provides some functionality specific to its context. This list is never exposed.

It appears to us that the explicit lists in `aglets` tend to be legacy code which could be replaced by priority queues and plain collections. Maintenance of the legacy portions of the code probably ought to start by replacing the explicit lists with general collections. Also, the list implementations are well-encapsulated.

Benchmark 4: `artofillusion`.

Finally, we examined the lists and some of the trees in the `aoi` ray tracer. We discuss the four linked lists and one of the five trees. These data structures account for 16 classes out of the 680 classes in this benchmark. (Here, we count the 8 identical list implementations in implementations of the `Distortion` interface as one list. They count separately in the Table.)

The `Distortion` interface specifies a `getPreviousDistortion()` method, so all 8 imple-

menting classes contain a private `previous` field. This implicit list cannot be manipulated, but it can be traversed by arbitrary callers.

Two other classes, `Instruction` and `ButtonStyle`, provide explicit lists. While `Instruction`'s `next` field is package-visible, both of the `next` fields are only accessed within those classes. In both cases, the use of an explicit list provides some short-term implementation advantages, but the use of a collection would not be much more onerous, and probably easier to maintain in the long term.

Finally, the `TriMeshSimplifier` uses fairly sophisticated list manipulations in its private `List` class, including appending a list to the end of another one. As with `lucene`, this algorithmically complicated class probably does need to use ad-hoc data structures. The fields which implement these data structures are only visible to the `TriMeshSimplifier`.

Summary of Data Structure Implementations.

Our exploration of a number of data structure implementations reveals a number of different reasons for developers to implement 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. Other lists were simply in old code, with no reason obvious to us for developers not to use system collections.

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.

Data structure implementations are exceedingly rare in our benchmarks; approximately 1 in 100 classes (in fact, 424 out of 42502 classes) implements an ad-hoc data structure. Our investigations suggest that even many of these implementations are superfluous or deprecated. While most of the data structure fields were only accessed locally, at least one `parent` field in our benchmark set was publically visible and accessed from other, unrelated, classes in the program.

Data Structure Fields.

To better understand how programs use data structures, and in particular which data structures programs use in practice, our tool also collects 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.

5.2 Data Structure Interfaces and Sharing

Our `DSFinder` tool also includes the static analysis from Section 4.3 for counting data structure interfaces and estimating the extent of data structure exposure. Table 1 also includes these data.

XXX include summary of those results.

We also manually investigated a number of benchmarks to classify their use of interfaces, starting with the `DSFinder` output. We describe some of our findings below.

Benchmark 1: `aglets`.

Recall that `aglets` contained 3 C-style lists. Our tool indicates that it also contains 21 data structure interfaces, of which it exposes 3. One example of a data structure interface is the `AgletEventListener`, which contains a package-visible `vector`, and a number of methods which manipulate the `vector`, such as `addMobilityListener()`:

```
public void addMobilityListener(MobilityListener l) {
  if (vector.contains(l)) {
    return;
  }
  vector.addElement(l);
}
```

This particular `vector` is fully-encapsulated, since no methods expose it.

We also investigated one of the exposed data structures, which occurs in some sample code, `SimpleMaster`. This class contains a private getter method (!), `getURLList()`. Such a method does not expose the data structure at all, and probably should be replaced with a simple field read. We did not construct our exposure analysis with such methods in mind.

Benchmark 2: `bcel`.

The Bytecode Engineering Library, `bcel`, has one linked list implementation, `InstructionList`. We therefore also studied it to see how many data structure interfaces it exports. This benchmark contains 14 data structure interfaces and exposes 3 data structures. One interesting example is the class `ClassGen`, which contains four `ArrayList` fields, one of which is `method_vec`, representing the methods in the class to be generated. Clients may (among other actions) add methods using the `addMethod()` class, query the list by calling `containsMethod()`, remove methods using the `removeMethod()` class, and request a copy of the list using `getMethods()`, which makes a copy of the method and returns that. This class therefore does not expose the `method_vec`.

Benchmark 3: `galleon`.

Finally, we discuss the `galleon` media server. This application contains no data structure implementations, but defines N data structure interfaces, of which M are exposed. We saw one example of an exposed data structure back in Figure 2: recall that the `GoBackConfiguration` class exposes its member `mPaths` list to its callers.

Qualitative Observations.

We conclude this section by presenting some qualitative observations on our benchmark set. Based on their extensive usage, developers clearly use system data structures and often delegate to them. Furthermore, we found that developers often explicitly document their design decisions when implementing ad-hoc data structures (and mark them with deprecated tags!). Developers most often documented ad-hoc data structures visible to external callers, and more rarely documented their decisions for private data structures.

Systems appear to be reasonably consistent in whether or not they expose data structures: either systems tended to expose data structures or not. Deviations from the norm are definitely worth flagging to the developer.

6. DISCUSSION

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

HYPOTHESIS 1. *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.

HYPOTHESIS 2. *Data structures are concentrated in a small portion of applications and libraries.*

Our data strongly support Hypothesis 2. 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 [24] 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.

7. THREATS TO VALIDITY

We survey threats to the validity of our hypotheses and discuss how we mitigate these threats.

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 [14] includes proprietary applications; these applications are generally similar to the open-source applications (although they do contain some outliers on some measurements).

Our corpus might not be representative because of biases in application domain; number and type of libraries used; and developer characteristics. 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 factors.

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 [19] 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.

8. 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.

8.1 Abstract Data Types & Encapsulation

Data abstraction, as first proposed by Liskov and Zilles in the context of operation clusters [12], 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 [22] 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[2]. Tempero's study of field use in Java [24] 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.

8.2 Empirical Studies

Researchers have recently conducted a number of studies of Java programs as-built. Collberg et al [6] presented quantitative infor-

[2]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.

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

Tempero [24] 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 2). Melton and Tempero also performed a related study [14] on cyclic dependencies between classes.

Malayeri and Aldrich [13] 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.

8.3 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 [15], 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 [20] and modular verification, as seen for instance in the Hob and Jahob analysis frameworks [11, 27].

9. 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 36 ad-hoc data structures.

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?

10. 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] S. J. Fink, R. Bodik, and M. Sridharan. Thin slicing. In *Programming Language Design and Implementation*, 2007.

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

[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] 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.

[13] 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.

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

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

[16] A. Nair. Object histories in java. Master's thesis, University of Waterloo, Waterloo, Ontario, May 2010.

[17] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, 2003.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] Sun Microsystems Inc. Collections framework overview. `http://java.sun.com/j2se/1.4.2/docs/guide/collections/overview.html`, 1999. Last accessed on 1 July 2010.

[24] 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.

[25] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Programming Language Design and Implementation*, pages 174–186, Toronto, Canada, June 2010.

[26] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Programming Language Design and Implementation*, pages 160–173, Toronto, Canada, June 2010.

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