Lucida: An Analyzing Program Browser

Derek Rayside, Lucy Mendel, Robert Seater and Daniel Jackson
MIT Computer Science and Artificial Intelligence Laboratory
{drayside, lmendel, rseater, dnj} @mit.edu
October 3, 2005

1 Introduction

We present Lucida, a program browser that highlights method calls that are more likely to be revealing of the design of the program under consideration. Our determination of which method calls are more interesting is made through a synthesis of ownership inference and mutability analysis.

Our criteria for identifying interesting method calls is outlined in Figure 1. If object a owns object b, then any call from a to b is less interesting — it is *ordinary*. It is expected and unsurprising for a whole to use and even mutate its parts. Similarly, we consider any call to a pure (ie, side-effect free, or non-mutating) method to be ordinary, whether the caller owns the callee or not. Such calls are less likely to reveal flaws or important aspects of the design. Calls to mutating methods where the caller does not own the callee are the most interesting — we name these calls *extraordinary*, because it is surprising to find an object mutating something that belongs to some other object.

Figure 1 Classifying calls with ownership and mutation

		Mutation	
		pure	**mutating**
Ownership	**owned**	ordinary	ordinary
	not-owned	ordinary	*extraordinary*

Extraordinary calls are not necessarily *erroneous*: they may be *essential* to the design of the program. However, extraordinary calls are never *unimportant* to the design of the program: they always reveal something interesting. Ordinary calls may be unimportant to the overall design of the program: for example, a call to a private helper method.

Lucida highlights the extraordinary calls, so that the programmer may investigate them and make a judgment as to whether they are erroneous or essential to the overall design of the program.

Previous program browsers have usually innovated in human-computer interaction while using a conventional syntactic analysis. Such an analysis approximates the source code, whereas Lucida endeavours to approximate the heap. An approximation of the heap is closer to what actually happens in the program at runtime, and so is more likely to reveal important design insights.

ETX'05 Eclipse Technology Exchange at OOPSLA'05.

1.1 Object Ownership

Object ownership is commonly defined as follows: an object x owns an object y if x dominates y in graph of referencing relationships in the heap: ie, x is the only object with a reference to y; there does not exist another object z that has a reference to y.

This common definition is useful for static type systems that provide guarantees about program properties [1; 5; 9; 12]. However, the common definition is less suited to revealing design insights about the program than it is to providing certain kinds of guarantees.

Consider the case where object y is referenced by both object x and object z. According to the common definition, neither could be the owner: y must be a globally visible object. This results in an ownership hierarchy that is wide and flat, with many objects at the top (global) level.

Our approach (which is similar to [8]) is to choose one of either x or z as the likely owner of y, and then highlight the edges from the non-owner as extraordinary. This approach to ownership results in a hierarchy that is deeper and has a lower branching factor, particularly at the top level.

Since our notion of ownership only needs to pick a likely owner for each object, and does not require a guarantee that only the owner references that object, we can use a relatively fast and lightweight (although still whole-program) analysis.

1.2 Mutation and Object Identity

Extraordinary method calls are those from non-owners to methods that mutate their receiver object. Identifying these calls requires knowing both the ownership structure and which methods are mutators. We perform a mutability analysis to partition methods into mutators and non-mutators.

We also partition fields into (immutable) *identity* fields and (mutable) stateful fields. Classes with only identity fields are sometimes referred to as value types in the literature. This partitioning allows us to distinguish methods such as hashCode(), which (should) only read identity fields. Calls to these methods are sometimes interesting and other times not, and Lucida gives the user an option whether to highlight them.

2 Analyses

Lucida is based on a synthesis of ownership inference and mutability analysis. Our current implementations of these analyses are relatively lightweight whole-program analyses. By 'lightweight' we mean not overly complicated and execute in a few minutes. A whole-program analysis is one that includes all of the relevant libraries that the program uses (eg, the JDK).

Call Graph Construction Object-oriented programs contain many method calls that are dynamically dispatched. Most high-level analyses, such as those we describe below, needs to have a reasonable call graph of the program. A static call graph construction algorithm resolves these call sites to a set of potential targets as a prelude to subsequent analyses. Lucida currently uses the *Rapid Type Analysis* (RTA) algorithm of Bacon & Sweeney [2; 3]. RTA prunes the results of a class hierarchy analysis by starting at the program entry points and removing from consideration classes that are never instantiated. RTA is roughly linear in the size of the program.

2.1 Ownership Inference

Our ownership inference algorithm infers a likely owner for each object. As with most points-to analyses, objects are identified by instantiation sites. Each live instantiation site in the program represents a potential *ownership context*, and is either owned by some other context or is shared (ie, owned by the root). Instantiation sites of the same class and with the same owner are coalesced. Ownership contexts are named proximately by their instantiated class and absolutely by their path in the inferred ownership hierarchy.

Let W (whole) be an ownership context, and let w be the class named proximately by context W. Let P (part) be an ownership context, and let p be the class named proximately by context P. We infer an ownership edge from W to P if any of the following criteria hold:

1. p extends w
2. w instantiates p and stores the result in a field $w.f$
3. w is the sole instantiator of p
4. a field $w.f$ is an *identity field* of w and has type p

The intuition for the first criterion is that if p extends w, then each object that instantiates p has a w object inside of it that it owns. In this sense our notion of ownership is closer to object layout than to traditional class diagrams.

Computing the second criterion precisely would require a flow- and context-sensitive analysis. We use a flow- and context-insensitive approximation that seems to work reasonably well in practice, although it is confounded by some creational design patterns such as factory.

The third criterion often holds in the case where p is an inner class of w, or w is an entry point to the program.

The inference of identity fields is discussed below in the subsection on mutability analysis.

2.2 Mutability Analysis

A mutability analysis determines which fields and objects are mutable, and which methods are mutating them. Lucida allows the user to filter out relationships based on their mutability characteristics. Our mutability analysis involves the following:

Immutable classes and identity fields. We consider all objects of class C to be immutable if all of the fields in C and all of C's super-classes are final and of an immutable type that has no mutable subtypes. This analysis requires iterating to a fixed-point.

A field $C.f$ is considered an *identity field* if it is final and immutable (and the type of f has no mutable subtypes). In other words, an immutable object is one in which all fields are identity fields. Identity fields are computed as part of the fixed point computation previously mentioned.

A *read identity* method is a special kind of pure method (ie, a method with no side effects). We consider that a method n is a read-identity method if it takes no parameters, reads only identity fields, and doesn't write any fields. Methods such as hashCode() often meet this criteria. Lucida allows the user to filter out calls to such methods.

Direct mutators. Method m directly mutates class C if m writes to C's field $C.f$. This analysis is a simple scan of the field writes of the program.

Indirect mutators. Method $D.n$ is considered an indirect mutator of class D if n instigates some method call chain that eventually writes to a field $E.g$, where E is owned by D (perhaps transitively). Computing indirect mutators requires knowing the ownership hierarchy, and also requires transitive closure, but does not require iterating to a fixed-point. Consider the following simple example:

```
class AddressBook {
    final Map m = new HashMap();
    void add(Person p, Address a) {
        m.put(p,a);
    }
}
```

We would infer that add() is an indirect mutator of AddressBook, because addEntry() calls put(), which eventually calls another method that writes into the field HashMapEntry.value, and that HashMapEntry is transitively owned by the AddressBook. Note that add() does not contain any field storing statements (ie, it is not a direct mutator of anything). Knowing that add() is an indirect mutator of AddressBook means that we can flag calls to add() from non-owners as extraordinary.

Representation exposure analysis. A method $C.m$ *directly exposes* the representation of C if it returns an alias to a non-identity field $C.f$ (Lucida actually uses a flow-insensitive approximation of this criterion). $C.m$ *indirectly exposes* the representation of C if it returns an alias it obtained from calling some other method n, where n exposes the representation of C or some other object that C owns. This analysis requires iterating to a fixed-point.

Method calls that expose the representation of the target object, like those that mutate the target object, are often interesting, and Lucida allows the programmer to filter the display on this basis.

3 Illustrative Example: Observer Design Pattern

The observer design pattern [7] provides a succinct illustration of how our concepts of ownership and extraordinary uses can illuminate the design of a program.

Consider the source code listed in Figure 2. The Main creates a ConcreteSubject and a ConcreteObserver, and then mutates the subject and prints out the observer. The observer registers itself with the subject, and observes the mutation.

Figure 3 shows a graph of the information inferred by our analyses. Nodes represent ownership contexts, with the name of their proximate class displayed. Solid edges represent ownership, and dotted edges represent extraordinary uses. One of these extraordinary uses is essential to the observer design pattern, and the other is a design error in this implementation. Our browser highlights these edges so that the programmer can make a judgment as to which is which.

Figure 4 is a screen shot of Lucida highlighting the extraordinary edge between ConcreteSubject and ConcreteObserver. This edge is an essential part of the observer design pattern: the observer must be registered with the subject (s.addObserver(this)).

Figure 5 is a screen shot of Lucida highlighting the extraordinary edge between ConcreteObserver and Date. This edge represents a design error, because Date is part of the mutable state of ConcreteSubject, and an alias to it is given to the ConcreteObserver. (Lucida correctly infers that java.util.Date is a mutable class.) This representation exposure makes it possible for the observer to directly mutate the internal state of the subject without the subject being aware of the mutation. If such a mutation were to occur, the subject would not be able to notify the other observers that the mutation had taken place, which defeats the whole purpose of the observer design pattern.

Recall that our notion of ownership is slightly different from the common definition. According to the common definition, Date is not owned by ConcreteSubject because of the alias from ConcreteObserver to Date. Instead, we choose ConcreteSubject as the likely owner of Date (using the criteria outlined in the previous section), and highlight the alias from ConcreteObserver as an extraordinary edge.

Comparison with conventional syntactic analysis. Most program browsers are based on a syntactic analysis of compilation dependencies. Such an analysis approximates the source code, whereas our analyses approximates the heap.

Figure 6 shows the information content that would be obtained by a conventional syntactic analysis, displayed in our browser for a fair comparison. Some program browsers are visually more sophisticated than ours (eg, [10; 11]), but they are typically used with a conventional syntactic analysis. These other browsers could also be reconfigured to display the results of our more sophisticated analyses.

Figure 6 is based on the package hierarchy, as is conventionally done, whereas Figures 4 and 5 are based on the inferred ownership hierarchy. The package hierarchy treats all classes, abstract classes, and interfaces within the observer package equally. The ownership hierarchy highlights the classes that are directly instantiated in the heap: Main, ConcreteObserver and ConcreteSubject. The ownership hierarchy also shows that ConcreteSubject comprises a Date and a LinkedList — information that is absent from the package hierarchy.

The conventional syntactic analysis also misses some important dependencies in the program because it does not consider subtyping nor dynamic dispatch. For example, Lucida identifies the edge from ConcreteSubject to ConcreteObserver.update() that would not be inferred by a conventional syntactic analysis.

Just as the package hierarchy treats all classes equally, the conventional syntactic analysis treats all method calls equally: there is no criteria to distinguish ordinary calls from extraordinary calls. This makes it more difficult for program browsers based on a conventional syntactic analysis to highlight those method calls that are more likely to reveal the design of the program.

4 Implementation

Lucida is implemented as an Eclipse plugin. The visualizer is our own SWT implementation of a hierarchical matrix browser. This style of information browser has previously been used for networked information spaces [15] and more recently for program browsing [14]. We experimented with the implementations from [15] and [14], but we were not able to customize them sufficiently for our purposes, and they were both based on Swing, which is less desirable than SWT for building an Eclipse plugin.

Lucida's analysis is built with the bytecode analyzer from [13] and the relational calculator from [4].

The entire analysis tends to take between 25 seconds and a few minutes, depending on the size of the program (3.6GHz P4). Recall that since this is a whole program analysis, even though the examples are small they require large parts of the JDK to be included in the analysis. This is not as fast as incremental compilation, but it is faster than many sophisticated whole-program analyses, and should be fast enough for regular usage.

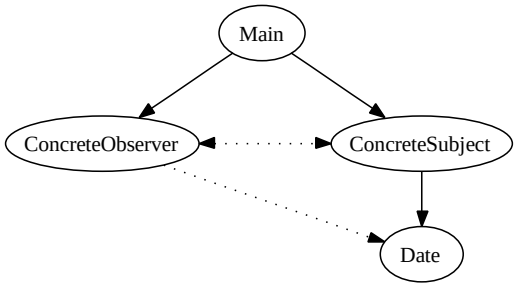
Figure 2 Source code for observer example

```
class Main {
    public static void main(String[] args) {
        ConcreteSubject s = new ConcreteSubject();
        ConcreteObserver o = new ConcreteObserver(s);
        s.setDate(1);
        System.out.println(o);
    }
}

abstract class Subject {
    private List observers = new LinkedList();
    void addObserver(Observer o) { observers.add(o); }
    protected void notifyObservers() {
        for (Iterator i = observers.iterator(); i.hasNext();) {
            ((Observer) i.next()).update(this);
        }
    }
}
class ConcreteSubject extends Subject {
    private final Date date = new Date();
    void setDate(long milli) {
        date.setTime(milli);
        notifyObservers();
    }
    Date getDate() {return date;}
}

interface Observer {
    void update(Subject s);
}
class ConcreteObserver implements Observer {
    private final ConcreteSubject s;
    ConcreteObserver(ConcreteSubject s) {
        this.s = s;
        s.addObserver(this);
    }
    void update(Subject s) {
        if (this.s == s) System.out.println(s.getDate());
    }
}
```

Figure 3 Ownership contrasted with extraordinary uses for observer design pattern example



Nodes represent ownership contexts, with the name of their proximate class displayed. Solid edges represent ownership, and dotted edges represent extraordinary uses. One of these extraordinary uses is essential to the observer design pattern, and the other is a design error in this implementation.

Figure 4 Extraordinary indirect mutation in observer: the observer registers itself with the subject. This is an essential characteristic of the observer design pattern.

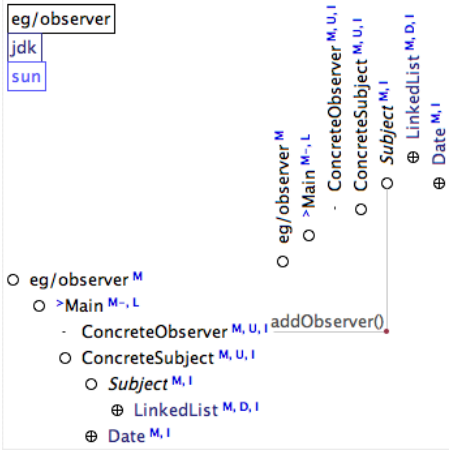


Figure 5 Extraordinary representation exposure in observer: the observer gets an alias to the subject's mutable state. This is a design error.

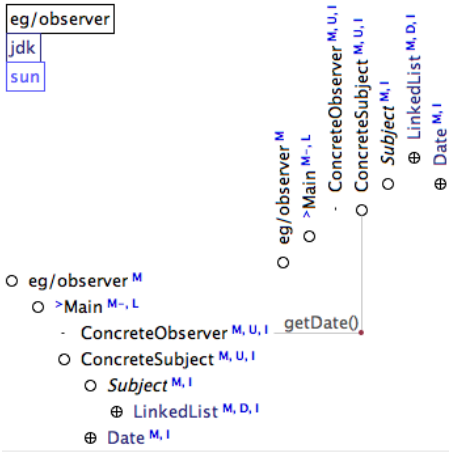
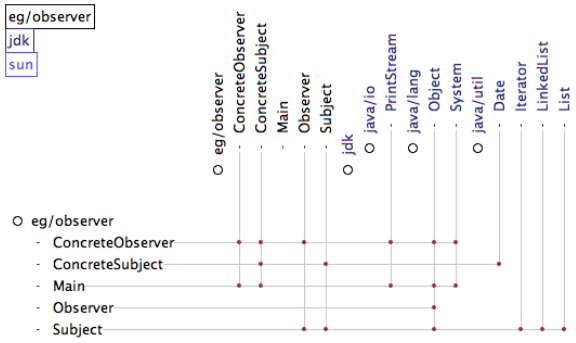


Figure 6 Observer example, conventional analysis



5 Discussion and Future Work

We have described Lucida, a program browser based on a synthesis of ownership inference and mutability analysis. These analyses serve to structure the information in the browser as well as to highlight parts of the program that are more likely to reveal the program's design: that is, method calls that cause mutation across the ownership hierarchy (the *extraordinary* calls).

We have introduced the concept of an *indirect mutator*: a method that indirectly causes mutation of some object that is transitively owned. (This idea was explained with a simple AddressBook example above.) This synthesis of ownership inference and mutability analysis enables us to identify methods as extraordinary that would otherwise be identified as ordinary, because they do not directly cause mutation. Such methods are often revealing of the design of the program.

The observer design pattern was used to illustrate how extraordinary method calls can be revealing of essential or erroneous aspects of the program's design, and to contrast the information displayed by Lucida with the information produced by a conventional syntactic analysis.

Our preliminary experience using Lucida (not previously discussed in this paper due to space constraints) suggests the following design observations:

- Programs designed around the concept of information hiding tend to have few extraordinary edges. Such programs are well modularized and contain few surprises.
- Well-designed programs tend to make systematic use of extraordinary edges (when such edges are used at all), whereas poorly designed programs seem to have ad-hoc and disorganized extraordinary edges.

We are currently working on the following:

- Experiments to validate our initial observations about the design insights gleaned from Lucida.
- Refining the analyses to deal with certain programming idioms, such as factories [7].
- Incorporating an escape analysis in the ownership inference to reduce the number of objects that we currently infer are 'shared'.
- Improving the user's experience of working with Lucida. The tool has some rough edges at the moment.

Acknowledgments

We thank Jonathan Edwards, Philip Guo, Patrick Lam and the referees for providing helpful comments on earlier drafts of this paper.

References

[1] Jonathan Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.

[2] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California at Berkeley, December 1997. UCB/CSD-98-1017.

[3] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In Coplien [6], pages 324 – 341.

[4] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient relational calculation for software analysis. *IEEE TSE*, 31(2):137–149, 2005.

[5] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT EECS, 2004.

[6] James Coplien, editor. *Proc.11th OOPSLA*, San Jose, CA, October 1996.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] David L. Heine and Monica S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In Rajiv Gupta, editor, *Proc.PLDI*, June 2003.

[9] Patrick Lam and Martin Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In Luca Cardelli, editor, *Proc.17th ECOOP*, volume 2743 of *LNCS*, pages 275–302, Darmstadt, Germany, July 2003. Springer-Verlag.

[10] Margaret-Anne Storey and Hausi A. Müller and Kenny Wong. Manipulating and Documenting Software Structures. In *Proc.ICSM*, pages 275–284, Opio (Nice), France, October 1995.

[11] Hausi A. Müller and K. Klashinsky. Rigi — A System for Programming-in-the-large. In *Proc.10th ICSE*, pages 80–86, Raffles City, Singapore, April 1988.

[12] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *Proc.12th ECOOP*, volume 1445 of *LNCS*, Brussels, Belgium, July 1998. Springer-Verlag.

[13] Derek Rayside. A generic worklist algorithm for graph reachability problems in program analysis. Master's thesis, University of Waterloo, 2001. Supervised by Kostas Kontogiannis.

[14] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using Dependency Models to Manage Complex Software Architecture. In Richard P. Gabriel, editor, *Proc.20th OOPSLA*, San Diego, CA, October 2005.

[15] Jürgen Ziegler, Christoph Kunz, and Veit Botsch. Matrix browser: visualizing and exploring large networked information spaces. In *SIGCHI extended abstracts*, pages 602–603, Minneapolis, MN, 2002.