The *Touchstone* interpreter evaluates programs in a language with an underlying model of computation including mutable data but excluding aliases. Using the language implemented in *Touchstone*, programmers can efficiently implement common programming abstractions, and programmers or compilers can efficiently, statically analyze programs to detect possible occurrences of unintentional object retention.

Categories and Subject Descriptors: D.1.0 [Programming Techniques]: General; D.1.5 [Programming Techniques]: Object-oriented Programming; D.1.m [Programming Techniques]: Miscellaneous; D.3.0 [Programming Languages]: General; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Design, Languages

Additional Key Words and Phrases: Unintentional object retention, observer design pattern, aliasing

1. MEMORY MANAGEMENT AND IMPERATIVE PROGRAMMING

In common examples of imperative or object-oriented programming languages, programmers can inadvertently subvert garbage collection by persistently aliasing memory. In his account of the design and implementation of Smalltalk-76 [Ingalls 1978], Ingalls describes the system as using reference counting to free programmers from the task of allocating and deallocating objects. Ingalls implies in his remarks on storage management that having the runtime system of a high-level language manage storage effectively frees programmers from having to reason about the state of heap memory. While observing that reference counting cannot reclaim objects participating in a cycle of references, Ingalls denies the importance of the problem pointing to the convenience of the object-oriented model for generating and sending messages through the cycle breaking the references and making the memory for the objects available to the system. Though Ingalls offers a resolution to the problem of cycles of references, he acknowledges that the resolution imposes on programmers a memory-management discipline of the kind that memory management by reference counting was supposed to eliminate, and in the end, Ingalls effectively concedes that programmers must reason about the state of the heap to correctly
write programs for Smalltalk-76.

In his justification for reference counting in Smalltalk-76, Ingalls reveals the hope-cum-understanding of many programmers. Programmers tend to think of garbage collection as freeing them from the burden of reasoning about the heap, but in fact, garbage collection by reference counting or any other mechanism only makes one, modest guarantee to programmers: the system will reclaim for reuse as convenient or necessary allocated but unreferenced memory.

In Structure and Interpretation of Computer Programs [Abelson and Sussman 1996], Abelson and Sussman betray an understanding of the ultimate aim of garbage collection as freeing programmers from having to reason about the heap, an understanding in common with the understanding of Ingalls, when they suggest that garbage collection is incompatible with imperative programming. Structure and Interpretation of Computer Programs introduces programming from a functional point of view where the understanding of Abelson and Sussman with regard to the aim of garbage collection as freeing programmers from the details of allocation and deallocation of objects conveniently coincides with the functional model. In functional programming, the model of programming admits no distinction between values, the results of computations, and memory cells, the physical locations storing data. Names bind to values rather than memory cells, and programs cannot mutate the value associated with a name. Of course, the interpreter of a functional programming language could execute on a machine with heap memory—in the interest of efficiency, the interpreter may mutate data and create and use aliases for memory while executing a functional program—but the functional model of programming conceals details of the machine and its memory organization safely behind the abstraction of the interpreter. In short, the hope for garbage collection and the behavior of garbage collection coincide in the functional model of programming because the programming model excludes the idea of a store of heap memory making irrelevant the need for programmers to allocate and deallocate objects.

The object-oriented model of programming descends from the imperative model of programming, and in his description of object-oriented programming and his implementation of an object-oriented model in Smalltalk-76, Ingalls appears to uncritically accept some assumptions of the imperative ancestors of his object-oriented model. In contrast to functional programming, the imperative model exposes programmers to a model of computation including a distinction between values and memory cells and assuming the existence of mutable data. Aliasing enters imperative programming as a consequence of computing with addresses, the identifiers for memory cells, which as explicit elements of the programming model invite treatment as data, and as expressed by Ingalls and his successors, the object-oriented model of programming as a descendent and heir of the imperative model of programming assumes aliasing as a support for the message-passing style. Aliasing memory makes the memory reachable from the registers of the virtual machine running a program and hides the memory from a garbage collector, and in common imperative and object-oriented programming languages where programmers can create aliases in any suitably typed variable in the text of the program, programmers can unintentionally lose track of references to memory protecting the memory from a garbage collector and effectively defeating garbage collection, a phenomenon known in the
Java programming community as unintentional object retention.

2. UNINTENTIONAL OBJECT RETENTION

Unintentional object retention poses serious problems for programmers—in addition to wasting memory, unintentionally retained objects can produce side-effects resulting in incorrect program behavior, behavior deviating from the specification of the program—and the subject of unintentional object retention generates a large body of literature and utilities devoted to reducing the workload associated with identifying and correcting the problem. On the IBM website [Goetz 2005], Brian Goetz of Quotix describes a common situation potentially resulting in unintentional object retention—unintentional object retention can occur when a programmer uses a hash table to associate meta-data with resources like sockets but forgets to remove resources from the hash table after finishing with the resources—and after presenting an example of unintentional socket retention, he writes, “In all but the most trivial cases, the techniques for identifying when the Socket becomes no longer used by the program resemble the annoying and error-prone techniques required for manual memory management.” In another article on the IBM website [Shirazi and Pepperdine 2003], Jack Shirazi and Kirk Pepperdine note that debugging heap problems can be so difficult that programmers sometimes convince themselves of the existence of errors in heap management in the virtual machine. Programmers’ problems with unintentional object retention suggest a need for academic study with the aim of finding a general solution to the problem.

2.1 An Observer Design Pattern Application

When a program describes a process creating multiple, contemporary instances of objects, in general connecting the instances to each other statically becomes impossible. Design Patterns: Elements of Reusable Object-Oriented Software [Gamma et al. 1995] describes an application of the observer design pattern to the task of changing the state of an unknown number of dependent objects in response to a change in an independent object, and the application of the observer design pattern illustrates another programming technique with the potential for unintentional object retention.

2.2 The Basic Observer Design Pattern Implementation

To take Java as an example, in an implementation of the observer design pattern, programmers typically create a Subject class corresponding to the subject in the design pattern. During the initialization of an instance of another object, an observer in the design pattern, the observer registers itself with the subject. Registration with the subject entails passing the subject a reference to the observer requesting registration. The subject stores the reference for the purpose of making callbacks to relay events of interest to the observer. Figure 1 on page 5 shows a minimal example of the observer design pattern including a Subject class representing the subject in the design pattern and an Observer class representing a generic observer in the design pattern. In the example, a user of the subject can send strings to the observers of the subject with the sendDatum method of the Subject class, which calls the sendDatum method of the Subject class to iterate over a vector of observers and call each observer’s receiveDatum method.
In the example, neither the Observer class nor the Subject class expresses any conditions for removing the instances of the Observer class from the vector of observers in the Subject class. The responsibility for removing references to the observers from the subject could belong to almost any other part of a program. Even when the specification for a program clearly assigns a class responsibility for managing references in the subject and the implementation of the program properly removes references to observers from the subject, the language offers programmers no contracts restricting the use of aliases; the code creating an observer instance, the code for the subject, or any other code in the program with an alias for an observer could pass a reference to the observer to another part of the program where the part of the program can create a persistent reference to the observer and protect the observer from garbage collection. In short, the simple example cannot suffice to draw conclusions about garbage collection in programs using the example’s classes.

2.3 An Observer Design Pattern Example with Output Side-Effects

2.3.1 A Java Implementation. The simple example of an observer in Figure 1 on page 5 does not represent a complete program and requires an exact context for understanding garbage collection of objects of the classes of the simple example. Witnessing unintentional object retention and its effects requires studying a complete example, and Figure 2 through Figure 5 contain a complete example of an implementation of the observer design pattern demonstrating unintentional object retention and its consequences in the presence of side-effects. Figure 2 on page 6 shows the implementation of the Subject class of a program. The Subject class implements the behavior of the subject of the design pattern. The Subject class, which only extends the Subject class of the simple example to follow coding conventions by adding interface information for the purpose of type checking, implements two interfaces, DatumListener and DatumSource, corresponding to messages sent in the program. In the sample program, the subject relays messages from a dedicated input object to dedicated output objects. Figure 3 on page 7 shows an implementation of a class for handling input from the system input device. By design, each input object serves one input device and creates an output object to process and display the input. Figure 4 on page 7 shows an implementation of a class for handling output by writing to the system output device. Finally, Figure 5 on page 8 shows the driver class, echo-memory-leak, for the sample program. To begin executing the program, the driver creates a subject and an input object. Appendix B on page App–3 contains the uninterrupted example of the Java program using the observer design pattern and having output side-effects. As in the simple example, programmers can pass references to observers to any part of the program and use the part of the program to persistently reference the observers and protect the observers from garbage collection, and the addition of type information in interfaces only restricts the types of variables permitted to hold the references to observers without imposing restrictions on the possible locations in the program declaring adequately typed variables.

2.3.2 A Scheme Implementation. Programmers can implement an analogous example in any language including imperative extensions sufficient to permit aliasing,
import java.util.*;

class Subject{
    Vector datumlisteners=new Vector();

    Subject(){};

    public void receiveDatum(String datum){
        this.sendDatum(datum);
    }

    public void sendDatum(String datum){
        Enumeration e=datumlisteners.elements();
        while(e.hasMoreElements()){
            Observer o=(Observer)e.nextElement();
            o.receiveDatum(datum);
        }
    }

    public void addDatumListener(Observer o){
        this.datumlisteners.addElement(o);
    }

    public void removeDatumListener(Observer o){
        while(this.datumlisteners.removeElement(o)){
        }
    }
}

class Observer{

    Observer(Subject sub){
        sub.addDatumListener(this);
    }

    public void receiveDatum(String datum){
    }
}

Fig. 1. A Simple Example of the Observer Design Pattern in Java

and readers can refer to Appendix C on page App–4 for the example’s implementation using the imperative extensions of the Scheme programming language.

2.4 Detecting and Correcting a Case of Unintentional Object Retention

In the example with output side-effects, the programmer likely intends that the instances of LocalOutObject die after processing output, but the program contains an error. The program never notifies the subject instance to remove references to observer instances, and the subject retains references to observers for the life of any process generated by the program. Before correcting a case of unintentional object retention, programmers must recognize the problem—in the example, the output side-effects simplify recognizing the existence of an error—and in the
Fig. 2. The Interfaces and Subject for an Observer Design Pattern Example in Java
class LocalInObject implements DatumSource{
    Subject subject;
    LocalInObject(Subject subject){
        this.subject=subject;
    }
    public void dataLoop(){
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        while(true){
            try{
                System.out.print("Enter a string: ");
                String datum=br.readLine();
                LocalOutObject outobj=new LocalOutObject(this.subject);
                this.sendDatum(datum);
            }catch(IOException e){}
        }
    }
    public void sendDatum(String datum){
        this.subject.receiveDatum(datum);
    }
}

Fig. 3. An Input Observer for an Observer Design Pattern Example in Java

class LocalOutObject implements DatumListener{
    Subject subject;
    String datum;
    boolean set=false;
    LocalOutObject(Subject subject){
        this.subject=subject;
        this.subject.addDatumListener(this);
    }
    public void receiveDatum(String datum){
        if(!this.set){
            this.set=true;
            this.datum=datum;
        }
        System.out.println(this.datum);
    }
}

Fig. 4. An Output Observer for an Observer Design Pattern Example in Java

public class echo {
    public static void main(String[] arguments) {
        Subject subject = new Subject();
        LocalInObject inobject = new LocalInObject(subject);
        inobject.dataLoop();
    }
}

Fig. 5. The Driver for the an Observer Design Pattern Example in Java

size and structure of the code permit isolating the problem rapidly and correcting the error entirely within either the LocalOutObject class or the LocalInObject class, which both have useful references to the unintentionally retained objects—for example, adding a this.subject.removeDatumListener(this) statement immediately after the System.out.println(this.datum) statement in the receiveDatum method of the LocalOutObject class corrects the mistake—but because in principle references can be copied to almost any location in a program, the process of discovering and correcting cases of unintentional object retention in a large, complicated program could involve examining numerous classes and tracing and modifying large, apparently distinct parts of the program. In short, recognizing cases of unintentional object retention and determining the bounds on the location in code of mislaid references quickly become difficult problems.

3. A SURVEY OF EXISTING AND SPECULATIVE RESPONSES TO UNINTENTIONAL OBJECT RETENTION

In attempting to solve the problem of unintentional object retention, language designers must first consider the possibility that the programming model and its languages contain abstractions sufficient to the development of techniques capable of implicitly handling the problem before considering modifying or extending a language or more radically modifying or deprecating and replacing the entire programming model. As unintentional object retention already poses a serious problem to programmers, literature specific to Java proposes several possible resolutions requiring evaluation, and the analysis of the causes of unintentional object retention presents the possibility of taking speculative approaches to the problem through programming model changes.

3.1 Weak References

One common, idiomatic approach to solving problems created by unintentional object retention as outlined by Goetz [Goetz 2005] uses weak references, references that the garbage collector will not follow in determining reachability. At first glance, weak references seem like a good approach to solving the problem, but on closer inspection, weak references do not appear to work in general. First, weak references only become useful once programmers trace the source of unintentional object retention in their programs. Programs in the imperative and object-oriented models of programming sometimes require persistent references to memory, so programmers cannot use weak references perversely. In short, programmers must analyze and
understand the evolution of the heap in their programs before deciding where to use weak references, and the requirement to understand the heap imposes on programmers a discipline almost as burdensome as manual memory management. Second, taking the discipline of reasoning about the heap to be an acceptable burden, weak references will not work in situations where an object has side-effects. Ensuring that an object no longer receives messages and participates in computations requires synchronously removing references to the object at the point in the program where the object should no longer participate in computations. In Goetz’s example with sockets, presumably the sockets are closed and no longer causing side-effects when unintentionally retained making weak-references an acceptable solution, but as a technique for making the memory associated with objects available for garbage collection, weak references represent references for all other purposes and fail to prevent the unwanted side-effects of unintentionally retained objects. Until the garbage collector removes an object available through a weak reference, the object continues to receive messages and might produce incorrect program behavior.

3.2 Programmer Managed Memory

Initially the inclusion of garbage collection in imperative and object-oriented languages might seem to deserve some blame for the prevalence of unintentional object retention, but in the final analysis, garbage collection only tempts programmers to neglect reasoning about the heap. Typically, garbage collectors function according to their contract to reclaim memory no longer accessible from the registers of the virtual machine of the runtime system, and programmers make the mistakes causing unintentional object retention.

Programmers’ misunderstanding of the contract of garbage collection and the temptation to ignore issues of heap management might recommend the removal of garbage collection from imperative and object-oriented languages. Removing garbage collection from a language leaves programmers to explicitly manage memory forcing programmers to constantly reason about the heap, but in programmer managed memory, programmers cannot misunderstand the memory-management contract or fall victim to the temptation to ignore the heap. Programmer managed memory has a simple contract: unless programmers explicitly deallocate allocated memory, the memory remains in use.

Before considering declaring garbage collection a failure in the imperative and object-oriented languages, the language designer needs to weigh the benefits of having the system manage memory against the cost of having programmers manage memory. Languages like C and C++ already put the burden for managing heap memory on programmers and give the language designer access to a long history of programming experience with imperative and object-oriented languages without garbage collection, and experience teaches that programmers commonly make mistakes in allocating and deallocating storage. Manually managed memory has the potential for unintentional object retention—a programmer who follows a conservative strategy for deallocating memory might err on the side of caution causing unintentional object retention—and in addition, manually managing memory creates another set of potential problems such as dangling references and freeing null references. The programming language community has judged from experience the problems of manually managing memory to be greater than the problems associated
with garbage collection, so removing garbage collection from a language inarguably represents a reverse in direction and betrayal of the lessons of experience.

### 3.3 Region-Based Memory Models

In *Cyclone* [Grossman et al. 2002], language designers find an alternative existing between manually managed memory and garbage-collected memory. *Cyclone* presents programmers a view of memory having both a garbage-collected heap and manually managed regions of memory allocated from the heap. *Cyclone* combines memory regions with a type system designed to statically detect a common error associated exclusively with manually managed memory, dangling references, and the designers of *Cyclone* prove the soundness of their type system meaning that in *Cyclone* attempts to dereference a dangling pointer become compile-time errors.

While *Cyclone* seems to promise the possibility of some relief from the problem of unintentional object retention, on closer examination, the relief requires programmers to identify suitable regions for each reference, in effect to undertake difficult reasoning about the heap. Depending upon the care exercised by programmers in choosing regions, rather than offering the best of both manually managed memory and garbage-collected memory, *Cyclone* could give programmers the worst of both worlds, and recognizing the sensitivity of the approach of *Cyclone* to the choices of programmers reveals a fundamental shortcoming of the idea of *Cyclone* as a response to the problem of unintentional object retention: if the problem of unintentional object retention reduces to errors in programmers’ attempts to reason about the heap, as in the case of the idea of manually managed memory, a solution to the problem cannot reduce to reasoning about the heap.

### 3.4 Restricted Aliasing

As the implementation of the observer design pattern examples in both *Java* and *Scheme* rely on aliasing to confound programmers’ expectations for garbage collection, unintentional object retention appears as a manifestation of aliasing more than as a consequence of the incompatibility of garbage collection with the imperative and object-oriented models of programming, and viewing unintentional object retention as a shortcoming of garbage collection in imperative and object-oriented languages treats aliasing as an indispensable feature of the imperative and object-oriented models of programming. In his paper, *On the Design of Programming Languages* [Wirth 1974], Wirth criticized including aliasing in its most general form in *Algol 68*. Wirth argued that his experience with aliasing in *Euler* showed programs with promiscuous aliasing to be difficult to understand. Unrestricted aliasing complicates reasoning about the heap, so unrestricted aliasing emerges as a leading cause of unintentional object retention. Taking the view that unintentional object retention traces principally to aliasing suggests reconsidering the role of aliasing in the imperative and object-oriented models, a project recommended by Wirth in his criticism of *Algol 68*.

A style of programming severely restricting or eliminating aliasing must necessarily prevent examples of unintentional object retention like the example of the implementation of the observer design pattern—eliminating aliasing limits the domain of references to a location in the heap memory to the scope of a single variable definition, and a program could not unintentionally retain allocated memory out-
side of the scope of the variable referencing the memory—but language designers must consider the broader consequences of a model of programming limiting or eliminating a pervasive part of extant languages in well known styles. Aliasing arises in imperative programming languages like C as a convenient means to improve efficiency and as a way to construct and modify dynamic, compound data structures such as lists, and taking for granted that efficiency does not have to suffer seriously from abstraction of the machine behind the interpreter or compiler of a language, an imperative language restricting aliasing must provide programmers a sufficiently rich set of dynamic, compound data structures as a means of combination of primitive elements of the language to ensure that programmers can build useful abstractions into their programs. In the object-oriented languages, aliases serve to support the message-passing style of programming by acting as handles for objects. For the object-oriented languages, a solution to the problem of unintentional object retention involving restricting or eliminating aliasing must provide another means for addressing messages to objects and defining relationships between objects.

4. THE TOUCHSTONE APPROACH

4.1 An Overview of the Touchstone Approach

As an experimental response to the problem of unintentional object retention, the Touchstone interpreter developed according to four principles:

—an imperative or object-oriented language must support mutable data;
—a language should either eliminate or severely restrict aliasing;
—a language restricting or eliminating aliasing should permit efficient implementation of common abstract data types;
—and, a language should concentrate expressions for the terminal conditions for an entity with the code for the entity.

In conventional imperative and object-oriented programming languages, the languages give programmers at least implicit, language-level access to the addresses of computational entities, and the models of evaluation behind the languages permit explicit manipulation of the addresses of computational entities as program data. By contrast, the languages offer no model notion of termination or language-level means for succinctly expressing the end of the computational life of an entity. When programmers explicitly manage memory by allocating and deallocating entities, language-level access to addresses as data combines with the exclusion of terminal conditions on entities from the programming model and the resulting inability to succinctly express at the language level the conditions on the termination of the life of an entity to lead to phenomena such as dangling references. When the runtime system of an imperative or object-oriented language manages memory for programmers, the combination leads to unintentional object retention.

The underlying model of evaluation of Touchstone recognizes the incongruity of permitting explicit modelling of the addresses of program entities as data while excluding support for modelling the termination of an entity, and the language of Touchstone reconciles the incongruent levels of modelling in conventional imperative and object-oriented languages by implicitly modelling the life of an entity to
provide language-level means to indicate the end of the computational life of the entity and by removing language-level support for aliasing and consequently the need for aliasing in the model of evaluation of the language. The model of evaluation behind the language of Touchstone uses a tree structure to relate entities in the process associated with a program, and the tree structure implicitly expresses the life of an entity and allows explicit, language-level expressions for the end of the computational life of an entity to map to model-level operations for pruning the tree.

The language of Touchstone descends from the object-oriented languages, but the language forbids aliasing for making programs difficult to understand and for causing unintentional object retention. The language of Touchstone makes irrelevant the problem of reasoning about the heap by eliminating aliasing and the means for making callbacks in favor of explicit, language-level primitives for message passing and by eliminating explicit allocation and deallocation of program entities in favor of well defined, language-level facilities for implicitly allocating and deallocating program entities. The language of Touchstone includes support for a useful, dynamic, compound data structure, the list, obviating the need for aliasing as a means to construct and modify dynamic, compound data structures.

4.2 A Note on Nomenclature

In conventional object-oriented languages, objects appear as machines having well defined interfaces, and keeping and using references to an object for the purpose of making callbacks reduces passing messages to pushing buttons to activate operations of the machine. Pushing buttons represents a degenerate form of passing messages and assumes knowledge by the operator of the operations of the machine. In effect, the degenerate form of message passing encouraged by aliasing subverts encapsulation. As biological entities respond to sensory messages without knowledge of interfaces of peers and completely encapsulate a view of the world, computational entities operating in a universe of computational entities without interfaces, without controls, naturally must completely encapsulate an epistemology, so the universe of biological entities admits a conceptual organization reflecting the goal of encapsulation in the design of Touchstone. To disassociate the language of Touchstone from the assumptions and the degenerate case of message passing in extant object-oriented languages, the built-in functions and special forms of the language of Touchstone and the nomenclature of its model of evaluation take inspiration from the behavior of biological entities. Programming in Touchstone involves defining hierarchies of species. Species behave like object factories in a traditional, object-oriented language; the species act as templates for creating agents, instances of computational entities, and define the initial responses of the agents to messages. An agent responds to a message using a message handler called an action.

5. PROGRAMMING FOR TOUCHSTONE

Figure 7 on page 15 and Figure 6 on page 14 show the parts of a complete stack implementation in the language of Touchstone. The language of Touchstone resembles object-oriented languages with species definitions corresponding to class definitions in other languages. A species definition can contain private data, and when Touchstone creates an agent, which corresponds to an object in object-oriented
languages, execution of the species body in the extended environment of the agent produces local copies of the private data of the species.

The parts in the figures can be concatenated to form a working stack implementation, or the definition of stack-element can be hidden within the definition of stack as in the example in Appendix D on page App–6. The block structure of the program defines the scope of the definition of a species. Taking the concatenation of the programs from the figures as an example, the example demonstrates the use of a large subset of special forms of the language of Touchstone omitting only the special forms accessing implicit sequencing—Appendix D on page App–6 contains the extended example using more built-in functions and special forms—so the example warrants some explanation. The example treats the stack as a container for a stack-element, and the container responds to the basic stack operations, push, peek, and pop. The stack requires no initialization.

On receiving a push message, the stack increments the variable storing the size of the stack and inserts its realm, a list of agents contained within the current agent, into a new stack-element; in the terminology of the language of Touchstone, the stack incubates its realm in a new stack-element. In effect, each push onto the stack packages the current top of the stack in a new stack element, which becomes the new top of the stack. By the design of the implementation of the stack, a stack or a stack-element can only contain one agent in its realm. The pop action uses two cases. In the first case, the stack could contain only one stack element, and having a stack-element consume its parent agent would destroy the stack agent, so instead the stack broadcasts the finish message to its realm, which causes any stack-element in the realm to invoke die and remove itself from the stack emptying the stack. In the second case, the stack contains more than one element, and the pop message causes the stack to send an advance message to the single stack-element in its realm, the top of the stack. The stack-element at the top of the stack responds to the advance message by sending an advance-aux message to the single stack-element in its realm, the second element of the stack. When the second element of the stack receives the advance-aux message, the stack-element calls the emerge special form and consumes the stack element containing it. In effect, the second element of the stack displaces the first element of the stack in the container for the stack. The pop operation decrements the size of the stack if the size of the stack is nonzero.

The peek action demonstrates communication from an agent to its parent agent. On receiving a peek message, the stack container sends a get-value message to any stack-element in its realm. When the stack-element at the top of the stack receives the get-value message, the stack element replies to the container by sending a result message with the react special form. The result message handler in the stack container passes the result message to its parent agent by in turn sending a result message. In the experimental implementation of Touchstone, if the parent of the stack container does not implement a result action, the runtime system stops with an error, but the internal representation of actions permits and anticipates modification of the implementation to discard messages having no handler in an recipient.

The tree structure imposed on the relationship between agents in the model of
Fig. 6. A Sample Stack Implementation for Touchstone

evaluation of the language of Touchstone and the language-level mechanisms for implicitly accessing and manipulating the tree structure support the construction of meaningful abstractions for data, and the example program using only the simple data and procedural abstractions unique to the language of Touchstone offers the implementation of a common abstract data type, the stack, as a demonstration of the usefulness of the unique aspects of the language. In addition, the explanation of the behavior of the implementation of the stack shows issues of efficiency in the execution of a program using the unique features of the language need only depend upon the efficiency of the implementation of the built-in functions and the special forms of the language.
(species stack-element

  (define value 0)

  (define init
    (action args
      (if (not (null? args))
        (set! value (car args))))))

  (define debug
    (action ()
      (begin
        (display value)
        (newline)
        (act (debug)))))

  (define get-value
    (action ()
      (apply react (list (list 'result value))))))

  (define advance-aux
    (action ()
      (emerge)))

  (define advance
    (action ()
      (act (advance-aux)))))

  (define finish
    (action () (die))))

Fig. 7. A Sample Stack Element Implementation for Touchstone

6. THE LANGUAGE

The Touchstone interpreter builds on a meta-circular evaluator for the Scheme programming language, and by design the language of Touchstone inherits its syntax, its primitive elements, and many of its basic, built-in functions and special forms from Scheme.

6.1 Interpreter Representations of Touchstone Abstractions

The interpreter internally represents a species as a tagged list including the text of the program in the species block and a reference to the environment of definition of the species. Touchstone uses species to construct computational entities termed agents. The interpreter internally represents an agent as a tagged list including a reference to the agent’s species, a reference to an environment constructed from the base environment of the agent’s species extended with definitions from the program of the agent’s species, a list of agents contained within the agent and termed a realm, and a reference to the parent agent of the agent. Touchstone defines the initial configuration of the interpreter as an agent with an empty species and an empty parent agent. Touchstone represents an action, a message handler associated with species, as a tagged list including the formal parameters of the action, the code for
the action, and a pointer to the environment of definition of the action.

6.2 The Model of Evaluation

When evaluating a Touchstone program, most program expressions modify the environment of an agent according to the environment model of evaluation, but Touchstone supports special forms for adding agents to the realm of the current agent. An agent may use special forms to generate messages for agents in its realm, for itself, or for its parent agent. Generating a message triggers changes in the environment and realm contexts of evaluation. Upon accepting a message from an agent, the interpreter loads the species, environment, realm, and parent agent of the target agent and evaluates the message. To evaluate a message, Touchstone resolves the name of the message in the environment of the agent, which initially contains a reference to an environment constructed from the environment of definition of the species of the agent extended with a single, empty frame of bindings, and applies the action bound to the message name to the arguments of the message. Touchstone supports special forms allowing an agent to remove itself from the realm of its parent agent or alter the hierarchy of realms by displacing its parent agent or encapsulating its realm in a new agent. Touchstone supports no special forms for aliasing.

6.3 Inherited Built-in Functions and Special Forms

Readers can consult the specification for Scheme [Abelson et al. 1998] for explanations of the built-in functions and special forms of the language. Implementation of the language of Touchstone as an extension to a meta-circular evaluator allowed omitting aliasing by specifically omitting support for the special form set-cdr!.

6.3.1 Built-in Functions. The language of Touchstone supports the following built-in functions from Scheme: exit, read, load, display, newline, list, cons, car, cdr, pair?, null?, make-vector, vector-length, vector-set!, vector-ref, vector?, eq?, eqv?, equal?, =, <, >, +, -, *, /, and not.

6.3.2 Special Forms. The language of Touchstone supports the following special forms from Scheme: define, set!, quote, and, or, if, cond, lambda, let, letrec, and begin.

6.4 Primitive Elements

The language of Touchstone inherits all of the primitive elements of Scheme.

6.5 Means of Combination

Like Scheme, the language of Touchstone supports function application as a means of combination of its primitive elements, and the language of Touchstone supports the list and vector from Scheme as a means of combination of primitive elements in compound data structures.

6.6 Means of Abstraction

Of course, the language of Touchstone supports the means of abstraction available in Scheme, data abstraction by define and procedural abstraction by the combination of define and lambda, and the language of Touchstone offers several additional means of abstraction in support of its metaphor.
6.6.1 **species**

— (species <species name> <species body>)

The `species` special form binds the `species name` to a structure containing a reference to the environment of definition of the species and the `species body`. The special form does not evaluate either of its arguments.

6.6.2 **action**

— (action <parameters> <action body>)

The `action` special form represents a message handler as a structure containing the `parameters`, the `action body`, and the environment of definition of the handler. The special form does not evaluate either of its arguments. The `action` special form is an analog of the `lambda` special form, and the `action` special form exists to allow the interpreter to distinguish message handlers from functions.

6.6.3 **conceive**

— (conceive <species name> ())
— (conceive <species name> () <message>)
— (conceive <species name> <agent name>)
— (conceive <species name> <agent name> <message>)

The `conceive` special form creates an instance of `species name` where `species name` resolves in the calling environment for `conceive`. The instance called an `agent` has a representation in the interpreter as a structure containing a reference to the species of the agent, an environment initially constructed as a reference to the environment of definition of the species extended by an empty frame of bindings and modified by executing the code of the species, an empty realm of agents, and a reference to the calling agent. The interpreter binds the `agent name` to the agent in the current realm, and to create anonymous agents, the `conceive` special form takes an empty list in place of the `agent name` argument. In a realm, the `agent name` need not uniquely bind to an agent; different instances of agents can share the same name effectively grouping agents in a realm. The `conceive` special form immediately sends the optional `message` to the agent after conception. The special form does not evaluate any of its arguments.

6.7 **Special Forms for Messaging**

The language of *Touchstone* offers three special forms in support of the message-passing style of programming.

6.7.1 **act**

— (act <message>)
— (act <agent name> <message>)

The `act` special form sends `message` to all agents named `agent name` in the current realm or to all agents in the current realm when the expression containing the special form omits the `agent name`. The special form does not evaluate any of its arguments.
6.7.2 react

- (react <message>)

The react special form sends message to the parent agent of the agent calling the special form. The special form does not evaluate its argument.

6.7.3 reflect

- (reflect <message>)

The reflect special form sends message to the calling agent. The special form does not evaluate its argument.

6.8 Special Forms Accessing Implicit Sequencing

To allow programmers to express notions of order to the interpreter for the purpose of optimization of the delivery of messages, the language of Touchstone offers four special forms giving programmers implicit access to the sequence of conception of agents within a realm.

6.8.1 conceive-first

- (conceive-first <species-name> ()
- (conceive-first <species-name> () <message>)
- (conceive-first <species-name> <agent-name>)
- (conceive-first <species-name> <agent-name> <message>)

The conceive-first special form behaves like the conceive special form except that conceive-first guarantees to make the agent conceived during its execution the chronologically first agent in the current realm. The special form does not evaluate any of its arguments.

6.8.2 conceive-last

- (conceive-last <species-name> ()
- (conceive-last <species-name> () <message>)
- (conceive-last <species-name> <agent-name>)
- (conceive-last <species-name> <agent-name> <message>)

The conceive-last special form behaves like the conceive special form except that conceive-last guarantees to make the agent conceived during its execution the chronologically last agent in the current realm. The special form does not evaluate any of its arguments.

6.8.3 act-first

- (act-first <message>)

The act-first special form behaves like the act special form except that act-first only sends message to the chronologically first agent in the current realm. The special form does not evaluate its argument.
6.8.4 act-last

— (act-last <message>)

The act-last special form behaves like the act special form except that act-last only sends message to the chronologically last agent in the current realm. The special form does not evaluate its argument.

6.9 Special Forms Altering Containment

The language of Touchstone offers two special forms supporting manipulation of the hierarchy of containment of agents within realms. The special forms exist to support nonlinear, recursive data structures such as trees.

6.9.1 emerge

— (emerge)

The emerge special form replaces the parent agent of the calling agent with the calling agent and updates the calling agent’s reference to its parent agent with a reference to the grandparent agent of the calling agent. The caller replaces its parent agent as the chronologically last member of the realm of its grandparent agent, and it joins the realm of its grandparent agent as an anonymous agent. The emerge special form has no effect when called from the default, top-level agent of the interpreter.

6.9.2 incubate

— (incubate <incubator_species> () )
— (incubate <incubator_species> () <message> )
— (incubate <incubator_species> <incubator_agent_name> )
— (incubate <incubator_species> <incubator_agent_name> <message> )

The incubate special form creates an agent of incubator_species, represented in the interpreter as a structure containing a reference to the species of the agent, an environment initially constructed as a reference to the environment of definition of the species extended by an empty frame of bindings modified by the program of the species, a reference to the realm of the caller, and a reference to the calling agent. The incubator_species resolves in the calling environment for incubate. The interpreter binds the non-unique incubator_agent_name to the agent in the current realm, and to create anonymous agents, the incubator special form takes an empty list in place of the incubator_agent_name argument. The incubate special form immediately sends the optional message to the agent after conception. The special form does not evaluate any of its arguments. The agent joins an empty realm, and the empty realm replaces the realm of the caller.

6.9.3 die

— (die)

The die special form removes the calling agent from the realm of its parent agent. When called from the default, top-level agent, the die special form causes the interpreter to exit.
6.10 Built-in Functions Turned Special Forms

6.10.1 apply

\[- (\text{apply} \ <\text{operator}> \ <\text{argument list}>)]\]

The \textit{apply} built-in function of \textit{Scheme} terminates evaluation with an error for special forms in the \textit{operator} position. Since the special forms of the language of \textit{Touchstone} do not evaluate their arguments, the language defines \textit{apply} as a special form accepting special forms as well as built-in and compound functions in the \textit{operator} position allowing programmers to invoke special forms of the language of \textit{Touchstone} with messages containing computed arguments.

7. THE IMPLEMENTATION OF THE INTERPRETER

\textit{Touchstone} runs under GNU/MIT Scheme 7.7.1 as an extension to a meta-circular \textit{Scheme} evaluator modelled on a meta-circular \textit{Scheme} evaluator in \textit{Structure and Interpretation of Computer Programs} [Abelson and Sussman 1996]. The extended meta-circular evaluator simulates only the environment and realm contexts of computation and implicitly uses the control structure of the underlying \textit{Scheme} implementation to return values from function calls rather than using a stack to explicitly simulate control.

The meta-circular evaluator implements environments as a list of frames where each frame contains a list of names and a list of corresponding values. Frames join the front of the list of frames, which represents the current frame, and bindings appearing earlier in the list shadow bindings appearing later in the list. The realm of an agent exists as a single frame associating a list of non-unique names with a list of corresponding agents. An agent joins a realm at the front of the list, the chronologically first position, making \textit{conceive-first} identical with \textit{conceive}.

The implementation of realms in the extended meta-circular evaluator uses the built-in lists of \textit{Scheme}. Access to the end of a list could require an amount of time increasing linearly in the size of the list making the \textit{conceive-last} and \textit{act-last} special forms require time increasing linearly in the size of the realm. In principle, an interpreter of the language of \textit{Touchstone} could internally represent realms with doubly linked lists and references to the ends of the lists making access to the end of a realm independent of the size of the realm, so the time requirements for realm processing represent a shortcoming of the implementation of \textit{Touchstone} rather than a fundamental flaw in the language.

In the experimental implementation, the extended meta-circular evaluator makes no distinction between interpreter entities created by \textit{action} and interpreter entities created by \textit{lambda}—the extended meta-circular evaluator will treat functions as message handlers—but the entities associated with \textit{action} and the entities associated with \textit{lambda} have different tags in the implementation permitting different treatment of the two types of entities when processing messages and anticipating extending the language to support designating private actions.

The entry point to the extended meta-circular evaluator, \textit{local-eval}, takes as arguments an expression, an environment, a species, a realm, and a parent agent. To the fullest extent possible, the extended meta-circular evaluator makes tail-recursive calls to \textit{local-eval} taking advantage of the tail-recursion optimization of
the Scheme standard [Abelson et al. 1998] and minimizing the stack requirements of the message-passing system.

The reader can download and run the source for Touchstone as described in Appendix A on page App–1.

8. STATIC DETECTION OF UNINTENTIONAL OBJECT RETENTION

The language of Touchstone offers a peculiar advantage in coding and static analysis over extant object-oriented languages: the language expresses at the language level the idea of the death of an agent and the end to an agent’s participation in computation, and the concise representation of the idea of an agent’s death in the die special form permits the interpreter of the language to completely dereference an object synchronously with execution at a clearly defined point in the program. The language of Touchstone makes no explicit reference to memory or memory management, and the implementation of the language can reclaim memory synchronously or asynchronously.

8.1 The Simplest Programs for Touchstone

A subset of the special forms of the language of Touchstone including incubate and die but excluding emerge and a restriction of the language to sending user-defined messages as opposed to arbitrary program expressions as messages suffices to conveniently express many, commonplace programming constructs as illustrated by the example of Figure 8 on page 22. The example shows an implementation of the specification implicit in the incorrect implementation of the observer design pattern example in Section 2.3.1 on page 4. For the common cases of programming, the example also highlights the intended benefit of the language of Touchstone: only an agent can end its computational life, and programmers and compilers can identify possible cases of unintentional object retention with the absence of any calls to die in the implementations of the actions for a species. While the existence of a call to the die special form in the implementation of an action of a species does not guarantee that the conditions necessary to terminate an agent of the species will ever be satisfied, the absence of a call to die in the implementations of the actions of a species does guarantee that an agent of the species recognizes no terminal conditions and persists for the remainder of the life of the process creating the agent. By contrast with the Java and Scheme implementations of the observer design pattern example, programmers can readily understand the implementation of the example in the language of Touchstone without tracing references and need only consider the code for an agent to formulate the terminal conditions of the agent.

8.2 Programs Using emerge

When a program uses the emerge special form, programmers and compilers can perform checks in addition to the checks for programs written with the subset of special forms excluding emerge and limited to sending user-defined messages, the simple programs for Touchstone. Programs using emerge admit a second means for ending an agent’s participation in computation: a species with an implementation calling emerge or containing a species with an implementation calling emerge defines a terminal condition for agents of the species—the stack-element species of
(species local-input-object
  (define ReadLoop
    (action ()
      (begin
        (display "Enter a number: ")
        (let ((datum (read)))
          (begin
            (conceive local-output-object locout)
            (apply act (list (list 'ReceiveDatum (list 'quote datum))))
            (reflect (ReadLoop))))))))

(species local-output-object
  (define local-datum false)
  (define ReceiveDatum
    (action (datum)
      (if (not local-datum)
        (begin
          (set! local-datum datum)
          (display local-datum)
          (newline)
          (die))
        (begin
          (display local-datum)
          (newline)
          (die))))))

(newline)
(conceive local-input-object locin (ReadLoop))

Fig. 8. The Analog for Touchstone of the Implementations of the Observer Design Pattern

the implementation of the stack abstract data type in Section 5 on page 12 stands as an example of a species using emerge to express a terminal condition—and the additional means of describing a terminal condition creates additional opportunities for analysis. Because an analysis for the case of simple programs identifies only die special forms with terminal points in agents, the analysis generates more conservative warnings than an analysis identifying emerge special forms with terminal points in agents, so programmers and compilers can safely apply the analysis for simple programs to programs using emerge.

8.3 Programs Using Special Forms as Messages

The design of the language of Touchstone does not anticipate that in everyday use solving common problems, programmers will resort to dynamically modifying the behavior of agents in a process and subverting attempts at statically understanding the program, but the language of Touchstone includes support for sending arbitrary expressions of the language to agents and dynamically overriding statically defined behaviors expressing the terminating conditions for an agent. Programs depending
upon agents dynamically altering themselves during processing defy static analysis, but the design of the language and model of evaluation of Touchstone aims not to limit programmers’ expressions but rather to provide a safe harbor to programmers in the common case of using expressions adhering to a programming discipline amenable to static analysis.

9. FUTURE WORK

As an experiment in programming language concepts, Touchstone and its language eschewed concerns for style in favor of easy implementation in an interpreter. In its current form, Touchstone represents a complete implementation of an interpreter for a full-featured language, but bringing the concepts in the language of Touchstone to a wider audience requires changes to the style of the language.

9.1 Separating Functions from Message Handlers

When processing messages, Touchstone makes no distinction between functions and message handlers, and improving the implementation should include extending the interpreter to take advantage of the tags distinguishing the internal representations of functions and message handlers and to dispatch messages only to message handlers.

9.2 Improving Syntax

The language of Touchstone adopted a syntax based on S-expressions because S-expressions represent data in Scheme and permit convenient treatment of Touchstone programs as data in an interpreter implemented in Scheme, but introducing the concepts of the language of Touchstone to a larger audience requires adopting a syntax resembling the syntax of mainstream languages like C, C++, and Java. Future work should include implementing the Touchstone model of programming in a language with a syntax similar to the syntax of Java.

9.3 Modifying the Set of Special Forms

The set of special forms of the language of Touchstone permits concise and efficient implementation of several common abstract data types, but extensive experience with programming in the language of Touchstone will likely reveal deficiencies in the language’s set of special forms and suggest convenient changes to the behavior of existing special forms. For example, the act special form evaluates none of its arguments, and experience suggests that changing act to evaluate its message argument might simplify the idiom for constructing messages. Future work should include developing extensive experience programming in the language of Touchstone for the purpose of analyzing and improving the fitness of each of the language’s special forms and with the intention of evaluating the completeness of the language’s set of special forms and extending the set as necessary.

9.4 Simulating Control and Supporting Parallel Processing

The implementation of Touchstone as an extension to a meta-circular evaluator for Scheme allowed the implementation to inherit and implicitly use part of the control structure of Scheme, specifically the control mechanism for returning values from calls to functions, but precluded simulating context switching between agents.
implementation in a simulator for a register machine allows explicit representation and modification of the control structure of a language in preparation to parallelize the model of evaluation of the language. The language of Touchstone offers an opportunity to extract implicit, language-level parallelism from programs at species boundaries, and future work should explore the potential of an interpreter of the language of Touchstone to automatically parallelize computations.

9.5 Formalization and Specification
Once the syntax and special forms of the language of Touchstone become firmly established, work on the language must include formalization of the grammar and the semantics of the language and the creation of a specification document and informal documentation on the use of the language including examples of the language’s common idioms and complete examples of programs in the language.

10. CONCLUSION
The observer design pattern and similar strategies for implementing applications in the imperative or object-oriented model in a language with garbage collection and aliasing become vulnerable to the problem of unintentional object retention for one, simple reason: in commonly used, garbage-collected imperative and object-oriented languages, programmers can only implicitly describe the termination of an object by removing all references to the object, but at nearly any location in the text of a program, the program can store references to an object. The languages give programmers explicit access to aliasing and only implicit mechanisms for destroying objects. The combination of explicit aliasing and explicit destruction as in languages like C and C++ succumbs to errors in heap-memory management like dangling references or attempting to free the same memory several times, problems worse than unintentional object retention, and remains unable to solve the problem of unintentional object retention. Touchstone takes a different approach to the problem of communication, which typically finds support in conventional, object-oriented languages through references, by design eliminating aliasing at the language-level and providing explicit, language-level support for indicating the end of the computational life of an object. In the design of the language of Touchstone, eliminating aliasing while providing explicit support for termination permits isolation of code related to the termination of objects to the object factories creating the objects and encourages in programmers the discipline of designing objects to recognize the conditions for their termination from their state data and received messages without regard to any external state. In short, Touchstone solves the problem of unintentional object retention by programming model changes and language-level devices intended to shift the focus of object-oriented development from the interconnection of objects in a process and to the protocol for communication between objects and the combination of the internal state of an object and the history of messages to the object defining the conditions for the object’s termination.

ACKNOWLEDGMENTS
I would like to acknowledge several professional and personal debts. Professionally, I would like to thank Dr. Todd Millstein for his patient advising and feedback as I

explored ideas to solve the problem of unintentional object retention. I would like to thank the members of my committee, Dr. Stott Parker and Dr. Majid Sarrafzadeh, for reading and reviewing my work. In addition, I would like to thank Dr. Alan Kay whose seminars at UCLA created the necessary environment for some of my exploration. Though I have never met them, I would like to thank Dr. Harold Abelson and Dr. Gerald Jay Sussman for their outstanding work in creating the didactic masterpiece in computer science, *Structure and Interpretation of Computer Programs*, the book giving me a coherent, complete view of the field and having the largest influence on the style and direction of my work.

Personally, I would like to thank my late mother, Michaelyn M. Noyes née Burke, whose enthusiasm for ideas and inquiring spirit live in her influence on me, and my step-father, Stephen B. Noyes, whose persistent and imaginative effort with me during my childhood kept me on an academic path. Finally, I would like to thank my friends in the UCHA cooperative, especially my friends in the Chinese community, who by their gentle concern and frequent inquiries regarding my progress helped me more than they will ever know.

REFERENCES


Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.


Received October 2008; revised November 2008; accepted November 2008
A. LINKS TO EXTERNAL MATERIAL

The reader may download the program for the Touchstone interpreter, the sample programs in the language of Touchstone, and the example programs in Scheme and Java. Section 7 on page 20 gives an overview of the development and testing environment for Touchstone, and the reader can expect all of the examples to execute in the development environment. The reader can load the interpreter from a running instance of Scheme using the load function, and the reader can load samples into Touchstone with Touchstone’s internal implementation of the load function. The files are available from the following locations:

— http://copper.chem.ucla.edu/~kulisics/txt/touchstone/echo_memory_leak.java

The link refers to the Java code demonstrating the observer design pattern with unintentional object retention, and the code contains as a comment a line correcting the error. The program complies with all Java standards and compiles and executes in the environment of the Sun Microsystems J2SE Developer’s Kit, version 1.6.0.07.

— http://copper.chem.ucla.edu/~kulisics/txt/touchstone/ObserverMemoryLeak.scheme

The link refers to the Scheme code demonstrating the observer design pattern with unintentional object retention, and the code contains as a comment a line correcting the error. The program runs under the GNU/MIT Scheme 7.7.1 interpreter.

— http://copper.chem.ucla.edu/~kulisics/txt/touchstone/touchstone.scheme

The link refers to the Scheme code for the Touchstone interpreter. The program runs under the GNU/MIT Scheme 7.7.1 interpreter.

— http://copper.chem.ucla.edu/~kulisics/txt/touchstone/touchstone-stack.scheme

The link refers to a program for a naïve implementation of a stack in the language of Touchstone. The naïve implementation uses a counter to record the size of
the stack. The program uses the counter to name stack elements, and to remove the top element from the stack, the stack agent addresses a message to the top element of the stack informing the element of the pop operation, which causes the element to issue a call to die. Since the language used in the program contains no notion of order, the most basic implementation of the interpreter must necessarily look through the list of all elements in the stack agent’s realm to find message recipients, and the complexity in time of the message dispatch grows linearly in the size of the stack. A sophisticated implementation of the interpreter could use a hash table to represent the realm achieving better performance in many cases, but the absence of any notion of sequencing at the language-level inspired the addition of special forms implicitly including agent sequencing and permitting implementation of more efficient stack operations.

http://copper.chem.ucla.edu/~kulisics/txt/touchstone/touchstone-stack-complex.scheme

The link refers to a program for a sophisticated implementation of a stack in the language of Touchstone. The sophisticated implementation uses special forms designed to manipulate the agent hierarchy running through the realms forming a part of the interpreter’s internal representation of an agent’s state. The program implements the push operation by calling Touchstone’s incubate special form to enclose the existing stack in a new stack element, and the program implements the pop operation by calling Touchstone’s emerge operation to allow the second element of the stack to consume and displace its parent agent, the first element of the stack, in the agent hierarchy.

http://copper.chem.ucla.edu/~kulisics/txt/touchstone/touchstone-queue.scheme

The link refers to a program for a queue implementation in the language of Touchstone. The implementation makes use of special forms designed to give explicit, language-level access to the order of conception of agents within a realm. With language-level references to the order of conception of agents within a realm, the interpreter can deliver messages to target agents in constant time.

http://copper.chem.ucla.edu/~kulisics/txt/touchstone/touchstone-observer-error.scheme

The link refers to a program for an implementation of the observer design pattern in the language of Touchstone. The implementation suffers from the problem of unintentional object retention, but unlike implementations in traditional imperative or object-oriented languages, the implementation in the language of Touchstone allows efficient examination of the program for possible instances of the error. For a particular species, in general programmers or compilers need only examine the actions of the species for the absence of calls to die and emerge and the actions of each enclosed species for the absence of calls to emerge to conclude that the species forms persistent agents.

http://copper.chem.ucla.edu/~kulisics/txt/touchstone/touchstone-observer-fixed.scheme

The link refers to a program for an implementation of the observer design pattern in the language of Touchstone. The example corrects the error of a previous sample by calls to die in the ReceiveDatum action of the local-output-object species.
B. A JAVA EXAMPLE OF UNINTENTIONAL OBJECT RETENTION

The Java example of the observer design pattern demonstrates unintentional object retention. The programmer probably intends to destroy each instance of a LocalOutObject when the instance completes its work, but the sample program retains a reference to the instance in the sole instance of the Subject class. The text of the program includes as a comment a line correcting the error.

```java
import java.io.*;
import java.util.*;

public class echo_memory_leak{
    public static void main(String[] arguments){
        Subject subject=new Subject();
        LocalInObject inobject=new LocalInObject(subject);
        inobject.dataLoop();
    }
}

class LocalInObject implements DatumSource{
    Subject subject;
    LocalInObject(Subject subject){
        this.subject=subject;
    }
    public void dataLoop(){
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        while(true){
            try{
                System.out.print("Enter a string: ");
                String datum=br.readLine();
                LocalOutObject outobj=new LocalOutObject(this.subject);
                this.sendDatum(datum);
            }catch(IOException e){}
        }
    }
    public void sendDatum(String datum){
        this.subject.receiveDatum(datum);
    }
}

class LocalOutObject implements DatumListener{
    Subject subject;
    String datum;
    boolean set=false;
    LocalOutObject(Subject subject){
        this.subject=subject;
        this.subject.addDatumListener(this);
    }
    public void receiveDatum(String datum){
        if(!this.set){
            this.set=true;
            this.d datum=datum;
        }
    }
}
```

System.out.println(this.datum);
// this.subject.removeDatumListener(this);
}
}

interface DatumListener{
    public void receiveDatum(String datum);
}

interface DatumSource{
    public void sendDatum(String datum);
}

class Subject implements DatumSource, DatumListener{
    Vector datumlisteners=new Vector();
    Subject(){
    }
    public void receiveDatum(String datum){
        this.sendDatum(datum);
    }
    public void sendDatum(String datum){
        Enumeration e=datumlisteners.elements();
        while(e.hasMoreElements()){
            DatumListener dl=(DatumListener)e.nextElement();
            dl.receiveDatum(datum);
        }
    }
    public void addDatumListener(DatumListener dl){
        this.datumlisteners.addElement(dl);
    }
    public void removeDatumListener(DatumListener dl){
        while(this.datumlisteners.removeElement(dl)){}
    }
}

C. A SCHEME EXAMPLE OF UNINTENTIONAL OBJECT RETENTION

The Scheme example of the observer design pattern, which uses the message-passing style in an analog of the Java example in Appendix B on page App–3, demonstrates unintentional object retention. The programmer probably intends to destroy each instance of a local-output-object when the instance completes its work, but the sample program retains a reference to the instance in the sole instance of the subject object factory. The text of the program includes as a comment a line correcting the error. The correction to the Scheme example takes a slightly different form than the correction to the Java example occurring by necessity in position not analogous to the Java example and illustrating the difficulty in isolating and correcting occurrences of unintentional object retention.

(define local-input-object (lambda (subject)
    (let ((SendDatum
        (lambda (datum)
            ((subject 'ReceiveDatum) datum))))
        (letrec ((ReadLoop
            (lambda ()
                (SendDatum
datum)))))
        (lambda ()
            (ReadLoop
datum)))))

(let ((outobj (local-output-object subject)))
  (begin
    (display "Enter a number: ")
    ((subject 'ReceiveDatum) (read))
    ;; ((subject 'RemoveDatumListener) outobj)
    (ReadLoop))))
(let ((dispatch
      (lambda (message)
        (cond ((equal? message 'SendDatum) SendDatum)
              ((equal? message 'ReadLoop) ReadLoop)
              (else (error "local-input-object: Unknown Message" message))))))
(dispatch)))))

(define local-output-object (lambda (subject)
  (let ((local-datum #f))
    (let ((ReceiveDatum
      (lambda (datum)
        (if (not local-datum)
            (begin
              (set! local-datum datum)
              (display local-datum)
              (newline))
            (begin
              (display local-datum)
              (newline))))))
      (let ((dispatch
        (lambda (message)
          (cond ((equal? message 'ReceiveDatum) ReceiveDatum)
                (else (error "local-output-object: Unknown Message" message))))))
      ((subject 'AddDatumListener) dispatch)))))

(define subject (lambda ()
  (let ((listeners '()))
    (let ((SendDatum
      (letrec ((SendIter
        (lambda (datum listeners)
          (if (null? listeners)
              #t
              (begin
                (((car listeners) 'ReceiveDatum) datum)
                (SendIter datum (cdr listeners)))))))
        (lambda (datum) (SendIter datum listeners)))))
      (let ((ReceiveDatum (lambda (datum) (SendDatum datum)))
          (AddDatumListener
            (lambda (object)
              (begin
                (set! listeners (cons object listeners))
                object)))
          (RemoveDatumListener
            (lambda (object)
              (begin
                (set! listeners (RemoveIter object listeners))
                object)))))
    (let ((dispatch
      (lambda (message)
        (cond ((equal? message 'ReceiveDatum) ReceiveDatum)
              ((equal? message 'SendDatum) SendDatum)
              ((equal? message 'AddDatumListener) AddDatumListener)
              ((equal? message 'RemoveDatumListener) RemoveDatumListener)
              (else (error "local-input-object: Unknown Message" message))))))
      ((subject 'AddDatumListener) dispatch)))))))
((equal? message 'DebugListeners) DebugListeners)
(else (error "subject: Unknown Message" message)))
)))))

(newline)
(define test (local-input-object (subject)))
((test 'ReadLoop))

D. AN EXAMPLE PROGRAM FOR TOUCHSTONE

The sample program uses the built-in functions and special forms of the language of Touchstone to define stack and queue species and construct instances of stack and queue. The init and init2 actions perform basic tests of the interfaces to the instances of the abstract data types.

(species stack
  (define size 0)
)

(species stack-element
  (define value 0)
)

(define init
  (action args
    (if (not (null? args))
      (set! value (car args)))))))

(define debug
  (action ()
    (begin
      (display value)
      (newline)
      (act (debug))))))

(define get-value
  (action ()
    (apply react (list (list 'result value))))))

(define advance-aux
  (action ()
    (emerge)))

(define advance
  (action ()
    (act (advance-aux))))

(define finish
  (action () (die))))

(define length
  (action () (apply react (list (list 'return size)))))))

(define debug
  (action () (act (debug))))))

(define peek
  (action ()
    (act (get-value))))

(define pop
  (action ()
    (if (= size 1)
      (begin
        (set! size 0)
        (reflect (peek))

(act (finish)))
(begin
  (if (> size 1)
    (set! size (- size 1))
    (set! size 0))
  (reflect (peek))
  (act (advance))))

(define push
  (action (val)
    (begin
      (set! size (+ size 1))
      (apply incubate (list 'stack-element '(') (list 'init (list 'quote val)))))

(define result
  (action (val)
    (apply react (list (list 'result val)))))

(define finish (action () (die)))

(define result
  (action (val)
    (begin
      (display val)
      (newline)))

(define init
  (action args
    (begin
      (conceive stack test)
      (act test (push 10))
      (act test (push 20))
      (act test (push 30))
      (act test (push 40))
      (act test (push 50))
      (act (debug))
      (act test (peek))
      (act test (pop))
      (act test (peek))
      (act test (pop))
      (act test (peek))
      (act test (pop))
      (act test (peek))
      (act test (pop))
      (act test (peek))
      (act test (pop))
      (act test (finish))))

(reflect (init)))

(species queue
  (define size 0)

  (species queue-element
    (define value 0)

    (define init
      (action args
        (if (not (null? args))
          (set! value (car args)))))

    (define debug
      (action ()

(begin
   (display value)
   (newline))))

(define get-value
   (action ()
      (apply react (list (list 'result value))))))

(define finish
   (action () (die))))

(define length
   (action () (apply react (list (list 'result size))))))

(define debug
   (action () (act (debug)))))

(define queue-insert!
   (action (val)
      (begin
         (set! size (+ size 1))
         (apply conceive-last (list 'queue-element '(' () (list 'init (list 'quote val)))))
   ))

(define queue-remove!
   (action ()
      (begin
         (if (> size 0)
            (set! size (- size 1))
            (set! size 0))
         (act-first (finish))))))

(define queue-front
   (action ()
      (act-first (get-value))))

(define result
   (action (val)
      (apply react (list (list 'result val))))))

(define finish (action () (die))))

(define init2
   (action args
      (begin
         (conceive queue test)
         (act test (queue-insert! 10))
         (act test (queue-insert! 20))
         (act test (queue-insert! 30))
         (act test (queue-insert! 40))
         (act test (queue-insert! 50))
         (act (debug))
         (act test (queue-front))
         (act test (queue-remove!))
         (act test (queue-front))
         (act test (queue-remove!))
         (act test (queue-front))
         (act test (queue-remove!))
         (act test (queue-front))
         (act test (queue-remove!))
         (act test (queue-front))
         (act test (queue-remove!))
   ))
(act test (finish)))
(reflect (init2))

E. THE OBSERVER DESIGN PATTERN IN TOUCHSTONE

The observer design pattern example in the language of Touchstone contains no special code to implement the behavior of the subject; the control structure of the interpreter contains the functionality of the subject in the observer design pattern. A programmer can readily determine by examining only the code of the local-output-object species that the instances of the local-output-object species die after processing a message. By contrast, determining that an observer instance dies in the Java and Scheme implementations requires examining every class receiving a reference to the observer.

(species local-input-object
  (define ReadLoop
    (action ()
      (begin
        (display "Enter a number: ")
        (let ((datum (read)))
          (begin
            (conceive local-output-object locout)
            (apply act (list (list 'ReceiveDatum (list 'quote datum))))
            (reflect (ReadLoop)))))

(species local-output-object
  (define local-datum false)
  (define ReceiveDatum
    (action (datum)
      (if (not local-datum)
        (begin
          (set! local-datum datum)
          (display local-datum)
          (newline)
          (die))
        (begin
          (display local-datum)
          (newline)
          (die))))))

(newline)
(conceive local-input-object locin (ReadLoop))