Eoops: An Object-Oriented Programming System for Emacs-Lisp

Chris Houser and Scott D. Kalter

chris@twinsun.com and sdk@twinsun.com

Twin Sun, Inc.
360 N. Sepulveda Blvd, Suite 2055
El Segundo, CA 90245-4462, USA

Background. Object-Oriented Programming (OOP) is a technique of modularizing programs into classes defining the structure and behavior of their instances or objects. OOP encourages the construction of large libraries of classes. The clients of a library use its classes as abstract data types; the implementors of a library use inheritance to factor and combine common code in the implementation. Detailed discussions of OOP and class libraries can be found in (Goldberg 83).

Abstract. Eoops (Emacs OOP System) implements a Smalltalk-80-like language in GNU Emacs-Lisp (Lewis 90). Eoops is a simple but efficient compiler for a class-based, single inheritance, object-oriented language that uses explicit message passing for both method invocation and state access.

This paper presents both a language description and a complete annotated implementation of Eoops. The reader of "The Eoops Language" section should be familiar with OOP, and ideally with Smalltalk-80. The reader of the "Implementation" section should be familiar with Common Lisp. Understanding some parts of the implementation requires knowledge of Emacs-Lisp.

1 The Eoops Language

Eoops supplies macros to create new classes, instantiate them, and send messages to instances. Eoops defines, not a complete programming language, but rather an object-oriented *extension* to a base language, here Emacs-Lisp. Thus, a programmer uses Eoops to specify classes and instances, but uses the base language to specify control and environment.

1.1 Global Macros

Eoops exports the macros `class`, `new`, and `$`, which may be used freely within Emacs-Lisp code. These macros are described below.

(`class` *class-name super* (*slot) *method**)**

Defines a new class called *class-name*. The symbol *super* names its superclass. Syntax:

$$\begin{array}{rcl}
\textit{slot} & \rightarrow & \textit{slot-name} \\
 & | & (\textit{slot-name documentation}) \\
\textit{method} & \rightarrow & ((\textit{selector parameter}*)\ \textit{form}*) \\
\textit{documentation} & \rightarrow & \textit{string} \\
\textit{class-name} & \rightarrow & \textit{symbol} \\
\textit{super} & \rightarrow & \textit{symbol} \\
\textit{slot-name} & \rightarrow & \textit{symbol} \\
\textit{selector} & \rightarrow & \textit{symbol} \\
\textit{parameter} & \rightarrow & \textit{symbol}
\end{array}$$

Each class introduces a namespace for slots and methods. Further, the names of classes form a namespace distinct from Emacs-Lisp's value, function, and property namespaces.

Example. A simple class of two-dimensional points:

```
(class point object (x y)
  ((print) ... )
  ((+ p) ... ))
```

This class, named `point`, is a subclass of the class `object`, has two slots named `x` and `y`, and methods (here with elided bodies) named `print` and `+`.

Definitions of OOP terminology:

Class. A class describes the structure (slots) and behavior (methods) of its instances. Each class is related to another class called its superclass. The inverse of the superclass relation is called subclass. Other languages call superclasses parent or base classes, and call subclasses child or derived classes.

Ancestors of a class. The ancestor relation is the reflexive transitive closure of the superclass relation. Thus, although a class cannot be its own superclass, a class is its own ancestor.

Descendants of a class. The descendant relation is the reflexive transitive closure of the subclass relation.

Class Tree. Since each class has a single superclass, but can be the superclass of several other classes, Eoops classes form a *tree* related by the superclass relation. The root of this tree is the class `object`.

Inheritance. The structure and behavior of a class is inherited by its subclasses. The subclass relation is also called the inheritance or inherits relation.

Instance. An instance of a class contains all the slots and methods defined in its class's ancestors.

Override. If like-named methods are defined both in a class `c` and in its descendant `d`, then `d`'s definition overrides `c`'s definition. This is analogous to variable scoping in blocks, if we view subclasses as nested within their superclasses.

The only thing one can do with a class is to instantiate it with the macro `new`:

(new *class-name*)

Creates and returns a new instance of the class named by *class-name.*

This new instance has private storage for each of its slots. An instance inherits its slots; it has a slot for each slot specified in its class's ancestors. In other languages, slots are called fields, data members, components, or instance variables.

This new instance has behavior as specified by its class's ancestors. An instance inherits its methods; it has a method for each method specified in its class's ancestors. In other languages, methods are called function members or operations.

After creating the new instance, `new` sends it an `init` message.

Example. Make `p` a new instance of the class `point`:

```
(setq p (new point))
```

The only things one can do with an instance are (1) pass it to functions, and (2) send it messages with the macro `$`:

($ *receiver selector argument**)

Sends *receiver,* which must be a value returned by `new`, the message consisting of *selector* and *arguments.* This searches the ancestors of *receiver*'s class, looking for a method named by *selector.* When a matching method is found, the evaluated *arguments* are bound to the the method's parameters, and the method's body is evaluated in this scope. The value of the `($ ...)` form is the value of the last form in the body. If no method matches, an error is signalled.

Example. Send `p` the message with selector `print` and no arguments:

```
($ p print)
```

`$` can be considered a generic function—a function whose behavior is a function of both the *selector* and the class of the *receiver.* `$` selects and invokes an appropriate *method.* Thus, methods describe the behavior of `$` for a particular selector and class of instances.

Slot Access. Besides the `methods` written by the programmer, each class has methods created by the Eoops compiler to read and write each slot. Sending these messages is the only way to access slots. For a slot named `s`, the reader method is named `s`; the writer method is named `s:`.

Example. Access `p`'s slots:

```
($ p x)  =>  nil    ; p's 'x' slot is initially NIL
($ p x: 5)  =>  5   ; set it to 5
($ p x)  =>  5      ; read the 'x' slot
```

The symbol `$` looks like "s," which is mnemonic for "send."

1.2 Within Method Bodies

Within method bodies, Eoops allows reading the variable `self`, using the macro `@`, and sending messages to `super`. These are described below.

self

Inside a method's body, the symbol `self` is bound to the receiver. The receiver is the object to which a message was sent invoking the currently-executing method. Methods can store `self` in variables and pass `self` to functions.

(@ *selector argument**)

Sends a message to `self`. This is just a shorthand for `($ self` *selector argument**`)`.

($ super *selector argument**)

Sends a message. Like `@`, the receiver is `self`, but unlike `@`, the method is that used when sending a message with the same *selector* to an instance of the super-class of the class in which the currently executing method is defined.

This is used when a programmer wants to make a slight change to an inherited method for selector `s`. The programmer writes a new method for `s`, that includes a call such as `($ super s)`.

Example. `subclass`'s `init` method initializes the slots declared by `subclass`, and then invokes `($ super init)` to ask `superclass` to initialize the slots declared by `superclass`:

```
(class subclass superclass (slot)
  ((init)
   (@ slot: 'initial-value)
   ($ super init))
  ... )
```

2 Example

This example, adapted from (McCall 80), demonstrates the syntax for Eoops' object-oriented features:

- class definition
- instantiating a class
- sending messages to self, super, and instances
- reading and writing slots using message passing
- subclassing with overriding and specialization

The class bank-account has superclass object, the single slot balance, and methods to initialize, deposit, and withdraw:

```
(class bank-account object (balance)
  ((init)
   (@ balance: 0))
  ((deposit: amount)
   (@ balance: (+ (@ balance) amount)))
  ((withdraw: amount)
   (and (<= amount (@ balance))
        (@ deposit: (- amount)))))
```

The withdraw: method returns the new balance, or nil if the account had insufficient funds.

Create an instance of class bank-account, call it b, and set its balance to $200:

```
(setq b (new bank-account))
($ b balance: 200)  =>  200
```

Exercise b by depositing and withdrawing:

```
($ b withdraw: 300)  =>  nil
($ b deposit: 150)  =>  350
($ b withdraw: 300)  =>  50
```

Next, the class overdraft-account is like its superclass bank-account, except that it adds the slot reserve, and overrides the method withdraw:. reserve is an account which is withdrawn from when the overdraft-account empties:

```
(class overdraft-account bank-account (reserve)
  ((withdraw: amount)
   (let ((m (min amount (@ balance))))
     (and ($ (@ reserve) withdraw: (- amount m))
          ($ super withdraw: m)))))
```

```
(setq o (new overdraft-account))
($ o reserve: b)
($ o deposit: 30)  =>  30
```

So now there's $30 in the overdraft account o, and $50 in the reserve account b. Withdrawing $45 takes $30 from o and the remaining $15 from b:

```
($ o withdraw: 45)  =>  0
($ o balance)  =>  0
($ ($ o reserve) balance)  =>  35
```

3 Rationale

This section explains design decisions made in the Eoops language and compares Eoops with other object-oriented languages.

Goals. We designed the Eoops language with the following (prioritized) goals:

- runs in a standard GNU Emacs
- executes quickly
- coexists with Emacs-Lisp code
- does not slow code not using Eoops
- is easy to remember and use
- supports debugging
- loads compiled files quickly
- compiles quickly
- compiles automatically
- runs using little space

Eoops meets these goals.

Comparison. Eoops is much simpler than other object-oriented languages embedded in Lisp. For example, Flavors (Moon 86) and CLOS have many additional features, such as generic functions, method combination, multi-methods, and multiple inheritance. Compared with Eoops, these languages are certainly more powerful and can express some things more concisely.

We chose Smalltalk-80's simpler model, rather than CLOS' complex model. We feel that Smalltalk's model is easier to understand, describe, and implement efficiently.

Rationale. Here we explain why Eoops:

- accesses slots by message sends
- is based on classes
- supports only single inheritance
- uses explicit *send* syntax
- is compiled

Slot Access. Eoops accesses slots by message sends. The Eoops compiler automatically generates methods to read and write slots. One advantage of this is that the Eoops programmer does not need to know the implementation of the instance's slots chosen by the Eoops compiler. Another advantage is that the caller can not distinguish between slot access and method invocation. This hides the callee's implementation of a selector, allowing changes to the selector's implementation (among various methods, slots, and even active values) without disturbing clients. SELF (Chambers 91) first implemented slot-access-by-method-passing. This equates client interfaces and subclass interfaces (Snyder 86).

Classes. Eoops offers classes and instances, rather than prototypes. Systems based on prototypes and delegation (Lieberman 86), such as SELF (Chambers

91), can be simpler, more flexible, and more concise. But to implement prototypes and delegation as efficiently as classes and instances requires a sophisticated compiler. We were familiar with simple implementation techniques. And classes have proved sufficient in 12 years of Smalltalk-80 use.

Single Inheritance. Eoops offers only single inheritance, rather than multiple inheritance. Multiple inheritance can provide greater expressive power, and is easily implemented. Again, however, we were familiar with simple implementation techniques and single inheritance proved sufficient in 12 years of Smalltalk-80 use. Smalltalk-80 offered optional multiple inheritance (Borning 82b) but it was seldom used. Ingalls (Ingalls 86) describes a manual technique for simulating multi-methods.

Explicit Sends. Eoops requires explicit *sends* using the `$` macro, rather than generic functions. CLOS' generic functions have many advantages because they are functions:

- they can be manipulated by both metaprogramming functions (e.g., `documentation`) and higher-order functions (e.g., `mapcar` and `some`).
- they abstract the distinction between methods and functions.
- they are terser (`(f x)` is shorter than `($ x f)`).
- they are more in the spirit of Lisp. CLOS' multimethods are a smooth generalization of Lisp's functions.
- they are easy to implement.

However, Eoops uses explicit *sends.* If instead Eoops used generic functions, then every call of a generic function would incur overhead, even when the first argument was not an instance. We wanted to avoid penalizing non-Eoops code.

Compiled. Eoops was first implemented as an interpreter. Its performance was adequate, but we wanted more speed. So we built a compiler, which now runs our benchmarks ten times faster than the interpreter. Eventually we discarded the interpreter.

The interpreter implemented classes and instances as association lists, and looked up slots and methods by linear search. Inheritance required a search up the class tree.

The compiler implements classes and instances as vectors, and looks up slots and methods in constant time. Inheritance is compiled out, by copying a superclass's slots and methods into its subclasses. Finally, the compiler uses Emacs' byte-compiler to macro-expand and compile method bodies.

4 Implementation

This section presents the annotated code[1] implementing Eoops. The annotations obey two conventions:

Argument names are in italic.

Packages are simulated by prefixing symbols with their file name and a colon. Eoops uses the three packages `eoops:`, `map:`, and `ce:`.

Readers familiar with backquote mechanisms in other Lisp dialects should note that the Emacs backquote syntax is different. Briefly, to translate between Emacs and more standard dialects, use `(' f)` → `'f`, `(, v)` → `,v`, and `(,@ x)` → `,@ x`.

4.1 Maps

The Eoops compiler represents classes as records or structures. These records, in turn, are implemented as *maps,* which are implemented here.

Maps implement mutable functions. Maps are often used in specification languages and very-high-level languages, such as VDL, Z, and Awk (where they are called associative arrays). They can be used as Smalltalk-like dictionaries (arrays where the key or index is generalized to an arbitrary object, rather than a small natural number), or C-like structures (mapping field names to field values).

Map operations construct maps, get the range, and get or set individual images.

`map:new` returns a new map. We implement a map as a pair, with head the symbol "map" and tail a Lisp association list. Here, an association list is a possibly empty list of lists, each with first element "argument" and second element "image." Each argument and image is an arbitrary Lisp object.

```
(defun map:new ()
  (list 'map))
```

`map:pairs` returns the association list of *map.*

```
(defun map:pairs (map)
  (cdr map))
```

`map:2nd` is just like Common-Lisp's `cadr`:

```
(defun map:2nd (list)
  (car (cdr list)))
```

`map:get` returns the image of *map* at *argument,* or `nil` if *argument* is not in the map's domain.

```
(defun map:get (map argument)
  (map:2nd (assoc argument (map:pairs map))))
```

`map:set` updates the value of *map* at *argument* to be *image* and returns the image.

```
(defun map:set (map argument image)
  (let ((pair (assoc argument (map:pairs map))))
```

[1]The source is copyrighted by Twin Sun Inc. See distribution for details.

```
    (if pair
        (setcar (cdr pair) image)
      (setcdr map (cons (list argument image)
                        (map:pairs map)))))
  image)
```

Finally, map:range returns the range of *map*, i.e., a list of the map's images.

```
(defun map:range (map)
  (mapcar 'map:2nd (map:pairs map)))
```

4.2 Utilities

Eoops uses three rather general utilities: symbol concatenation, a primitive LOOP, and a destructuring LET.

eoops:symbol-concat returns a symbol whose printstring is the concatenation of the list of *symbols/strings*.

```
(defun eoops:symbol-concat (&rest s)
  (let ((string '(lambda (x) (format "%s" x))))
    (intern (apply 'concat (mapcar string s)))))
```

ce:for binds *variable* to successive car's of *list* and evaluates *body*, returning the list of results. This is just a convenient abbreviation of mapcar. The 'ce:' prefix denotes our library of Common Emacs-lisp functions and macros.

```
(defmacro ce:for (symbol list &rest body)
  (` (mapcar '(lambda ((, symbol)) (,@ body))
          (, list))))
```

ce:let is a destructuring let. (let *bindings* . *body*) binds *bindings*, then evaluates forms in *body*, returning the value of the last form. *bindings* is a list; each element is either a symbol bound to nil or a list (*pattern value*) binding the symbols in *pattern* to corresponding *values*. Each *value* can refer to symbols already bound in *bindings*. For example:

```
  (ce:let (((first second . tail) '(1 2 3 4)))
     (list first second tail))
  =>  (1 2 (3 4)).
```

The implementation uses ce:bindings:

```
(defmacro ce:let (bindings &rest body)
  (` (let* (, (ce:bindings bindings)) (,@ body))))
```

ce:bindings, given *pattern*, returns a list of bindings. *pattern* is a tree of symbols. ce:bindings returns a list ((*symbol access*)*) where *symbol* occurs in the *pattern* and *access* is an accessing expression of nested calls to car and cdr.

```
(defun ce:bindings (bindings)
  (let ((s 'ce:let)
        (b '(lambda (pattern path)
              (cond
               ((null pattern)
                nil)
               ((symbolp pattern)
                (list (list pattern path)))
               ((consp pattern)
                (append
                 (funcall b (car pattern)
                            (list 'car path))
                 (funcall b (cdr pattern)
                            (list 'cdr path)))))))
    (apply
      'append
      (ce:for bind bindings
        (if (and (consp bind) (consp (car bind)))
            (cons (list s (nth 1 bind))
                  (funcall b (car bind) s))
          (list bind))))))
```

4.3 Eoops

Eoops requires two Emacs-Lisp libraries: the Emacs byte-code compiler bytecomp, and Emacs' pseudo backquote. Unfortunately, bytecomp neglects to provide, so we must load it with load-library.

```
(provide 'eoops)
(load-library "bytecomp")
(require 'backquote)
```

Eoops automatically recompiles object files when their source files change. To do this, Eoops needs to be able to find the source files. We assume that all Eoops source files are in the directory named by the variable eoops:class-path. A class named foo is defined in the file foo.el in this directory.

```
(defvar eoops:class-path "/local/emacs/elisp/classes")
```

All the classes are kept in a map bound to the global variable eoops:classes. This maps from class names to class records.

```
(defvar eoops:classes (map:new))
```

4.4 Public Macros

Here are the macros to be used by the Eoops programmer.

To get the code to fit in this journal's two column format, we have used several abbreviations. These symbols are used throughout the code:

c	class
p	parent of class c
nc	name of c
np	name of p
s	subclasses of class c
cv	compiled class vector
pv	compiled parent
f	compiled method (function)

The class macro creates a new class record, fills in its fields, and then compiles the class. The class record remembers the class's name, slots, methods, and other information.

eoops:compile-class, below, implements inheritance by copying information from superclasses to subclasses. Thus, subclasses depend on their superclasses; if the superclass changes, the subclass should be recompiled. To implement this, each class remembers it subclasses in the field called children.

```
(defmacro class (nc np slots &rest methods)
  (let ((c (or (map:get eoops:classes nc)
               (map:new))))
    (let* ((np (map:get c 'parent))
           (p  (map:get eoops:classes np))
           (s  (map:get p 'children)))
      (if p (map:set p 'children (delq nc s))))
    (eoops:require-class np)
    (let* ((p (map:get eoops:classes np))
           (s (map:get p 'children)))
      (if p (map:set p 'children (cons nc s))))
    (map:set c 'parent np)
    (map:set c 'children (map:get c 'children))
    (map:set c 'slots slots)
    (map:set c 'name nc)
    (map:set c 'compiled nil)
    (if (stringp (car methods))
        (setq methods (cdr methods)))
    (map:set c 'methods methods)
    (map:set eoops:classes nc c)
    (eoops:compile-class c)
    (map:set c 'modtime
             (eoops:class-mod-time nc 'obj)))
  (list 'quote nc))
```

The new macro just calls the function eoops:new after quoting the *class* argument:

```
(defmacro new (nc)
  (` (eoops:new (quote (, nc)))))
```

The @ macro is just a shorthand:

```
(defmacro @ (&rest arguments)
  (` ($ self (,@ arguments))))
```

$ implements fast message passing. The $ macro implements a partial method-lookup at compile-time, by precomputing the hash code for the *selector*. Then $ constructs code to dispatch on the class of the *receiver* at run-time. There are three cases:

(1) In the general case, ($ r s . a) compiles to

```
(let ((eoops:receiver r))
   (funcall
     (cdr (assq s (aref (aref eoops:receiver 0)
                        h)))
     eoops:receiver . a))
```

where h is the hash code of the selector s. Instances are represented as vectors, the first element of which is their class. So (aref eoops:receiver 0) is the class of the receiver. Compiled classes are represented as vectors. The first two elements of these vectors give the class's name and instance size. The remaining elements implement a hash table, mapping selectors to methods. (aref ... h) finds the selector's hash bucket, (assq s ...) finds the entry, and (cdr ...) finds the method. Finally, the method is funcall'ed on the receiver and the arguments.

(2) In the special case where the receiver is a symbol, evaluating it is cheap and can cause no side-effects, so we can dispense with the let and compile the simpler

```
(funcall (cdr (assq s (aref (aref r 0) h))) r . a)
```

(3) In the special case where the receiver is the symbol super, its class is known at compile-time. The class is the value of the symbol eoops:super-class, which is bound by eoops:compile-class and accessed here via dynamic binding. The compiled code is complex, since it looks up the class at run-time, so for a class whose parent is named *bar* the run-time code is:

```
(let ((super (map:get (map:get eoops:classes
                               'bar)
                      'compiled)))
  (funcall (cdr (assq s (aref super h)))
           self
           . a))
```

Notice that the receiver is self, but the method is sought in the class bar which may have actually inherited the desired method from its ancestors.

The $ macro is implemented as:

```
(defmacro $ (receiver selector &rest args)
  (let ((h (eoops:hash selector))
        (s (list 'quote selector)))
    (cond
     ((eq receiver 'super)
      (` (let ((super
                (map:get (map:get
                          eoops:classes
                          '(, eoops:super-class))
                         'compiled)))
           (funcall (cdr (assq '(, selector)
                               (aref super (, h))))
                    self
                    (,@ args)))))
     ((symbolp receiver)
      (` (funcall
          (cdr (assq (, s) (aref
                            (aref (, receiver) 0)
                            (, h))))
          (, receiver)
          (,@ args))))
     (t (` (let ((eoops:receiver (, receiver)))
             ($ eoops:receiver
                (, selector)
                (,@ args))))))))
```

4.5 Compilation

Compilation of a class record is performed by eoops:compile-class. The compilation of a class is comprised of the following steps:

(1) Create a new class vector, *cv*.

(2) Store the name of the class, *nc* in *cv*.

(3) Store each user-defined method in *cv*. The function eoops:store-method finds the correct hash position and byte-compiles the method body. Note that

the class and selector names are prepended to the documentation string to make an emacs debugger stack trace somewhat more readable.

(4) Create and store the methods for slot access.

(5) Write the class record to disk with the *cv* containing only those methods defined by the class itself.

(6) Make a new *cv* that inherits behavior from *pcv*.

(7) Recompile any subclasses that have already been loaded. This should only occur during class development when a class with children is being *reloaded*.

After compilation, a loadable version of the compiled class record exists on disk, the compiled class is resident in memory, and the class's children, if any, have been updated.

```
(defun eoops:compile-class (c)
  (let* ((np (map:get c 'parent))
         (p  (map:get eoops:classes np))
         (pcv (map:get p 'compiled))
         (nc (map:get c 'name))
         (cv (make-vector eoops:vtable-size nil)))
    (aset cv 0 nc)
    (let ((methods (map:get c 'methods))
          (eoops:super-class np))
      (ce:for method methods
        (ce:let ((((selector . parms) . body) method)
                 (doc
                  (format "(%s %s) " nc selector))
                 (body
                  (if (stringp (car body))
                      (cons (concat doc (car body))
                            (cdr body))
                    (cons doc body)))
                 (code
                  (` (lambda (, parms) (,@ body)))))
          (message "Compiling %s %s..." nc selector)
          (eoops:store-method cv selector code))))
    (let ((i (if pcv (aref pcv 1) 1))
          (slots (map:get c 'slots)))
      (aset cv 1 (+ i (length slots)))
      (ce:for slot slots
        (if (consp slot) (setq slot (car slot)))
        (eoops:store-method
         cv slot (` (lambda () (aref self (, i)))))
        (eoops:store-method
         cv
         (eoops:symbol-concat slot ":")
         (` (lambda (v) (aset self (, i) v))))
        (setq i (1+ i))))
    (map:set c 'compiled cv)
    (eoops:write-class c)
    (let ((pv (map:get p 'compiled)))
      (map:set c 'compiled
               (eoops:inherit-behavior pv cv)))
    (mapcar '(lambda (ns) (eoops:compile-class
                           (map:get eoops:classes ns)))
            (map:get c 'children))))
```

4.6 New

eoops:new checks, at run-time, that the specified class, *nc*, is loaded. If it is not currently loaded, eoops:require-class is invoked to load it. Normally, a class is repeatedly instantiated but only the *first* invocation of eoops:new may require the expense of loading the class. After the class record, *c*, is retrieved, the *cv* is retrieved, the instance vector is created and initialized, an init message is sent to the new instance, and finally the new initialized instance is returned.

```
(defun eoops:new (nc)
  (let* ((c (or (map:get eoops:classes nc)
                (progn (eoops:require-class nc)
                       (map:get eoops:classes nc))))
         (cv (map:get c 'compiled))
         (self (make-vector (aref cv 1) nil)))
    (aset self 0 cv)
    ($ self init)
    self))
```

4.7 Class Vectors and Hashing

The first two elements of a class vector are the class's name and instance vector size. The rest of the class vector elements implement a hash table for the methods for the selectors to which the class responds. The size of the hash table is stored in the constant *htable-size*. Therefore, the length of a class vector, *vtable-size*, is 2 + *htable-size*.

```
(defconst eoops:htable-size 23)
(defconst eoops:vtable-size (+ 2 eoops:htable-size))
```

eoops:hash returns a relatively unique integer for *symbol*, between 2 and *htable-size*.

```
(defun eoops:hash (symbol)
  (let* ((s (symbol-name symbol))
         (i (length s))
         (r 0))
    (while (< 0 i)
      (setq i (1- i))
      (setq r (+ r r (aref s i))))
    (+ 2 (% (max r (- r)) eoops:htable-size))))
```

eoops:store-method byte-compiles and stores *method* in *cv*'s hash bucket for *selector*.

```
(defun eoops:store-method (cv selector method)
  (let* ((f (byte-compile-lambda
             (eoops:add-self method)))
         (h (eoops:hash selector))
         (bucket (aref cv h)))
    (aset cv h
      (eoops:update-cv-bucket bucket selector f))))
```

eoops:add-self, given a lambda expression *f*, prepends the symbol self to the arguments in *f*. This is required since $ calls a method with the receiver as an additional argument prepended to those specified in a method definition.

```
(defun eoops:add-self (f)
  (ce:let (((lambda arguments . body) f))
    (` (lambda (self (,@ arguments)) (,@ body)))))
```

`eoops:inherit-behavior` copies the methods from the parent-vector to the class-vector. Called when the compiled code is loaded, this implements load-time inheritance.

```
(defun eoops:inherit-behavior (pv cv)
  (let ((ncv (make-vector eoops:vtable-size nil))
        (i eoops:vtable-size))
    (aset ncv 0 (aref cv 0))
    (aset ncv 1 (aref cv 1))
    (while (< 2 i)
      (setq i (1- i))
      (if pv (aset ncv i (copy-alist (aref pv i))))
      (ce:for entry (aref cv i)
        (aset ncv i (eoops:update-cv-bucket
                      (aref ncv i)
                      (car entry)
                      (cdr entry)))))
    ncv))
```

`eoops:update-cv-bucket` updates the hash table *bucket*'s alist to contain the lambda expression *f* for index *selector*.

```
(defun eoops:update-cv-bucket (bucket selector f)
  (let ((pair (assq selector bucket)))
    (cond (pair (rplacd pair f) bucket)
          (t (cons (cons selector f) bucket)))))
```

4.8 Storing and Loading Compiled Classes on Disk

`eoops:require-class` loads the specified class, *nc*, when either the class has never been loaded or when its sources, if available, are newer than the object file.

```
(defun eoops:require-class (nc)
  (let* ((s-time (eoops:class-mod-time nc 'src))
         (o-time (eoops:class-mod-time nc 'obj))
         (which
          (cond
           ((not nc) nil)
           ((and o-time s-time
                 (eoops:time-newer o-time s-time))
            'obj)
           ((and o-time (not s-time)) 'obj)
           (t 'src))))
    (cond
     ((not nc) t)
     ((and (map:get eoops:classes nc)
           (eq which 'obj))
      t)
     (t (eoops:load-file nc which)
        (let* ((c (map:get eoops:classes nc))
               (np (map:get c 'parent))
               (p (map:get eoops:classes np))
               (c-time (map:get c 'modtime))
               (p-time (map:get p 'modtime)))
          (if (and p (eoops:time-newer p-time c-time))
              (eoops:load-file nc 'src)))))))
```

`eoops:load-file` loads the class, *nc*, from disk. *type* indicates whether to load the `.el` or the `.elc` file.

```
(defun eoops:load-file (nc type)
  (load-file (eoops:class-file-name nc type)))
```

`eoops:load-class` stores the class record *c* in `eoops:classes` under the name *nc*. This function is explicitly invoked in a class's object file. Loading a class enforces that the parent is loaded. The parent's class vector is then copied and merged with the current class's class vector by `eoops:inherit-behavior`. This inheritance step is done at load time so that a new object file need not be created and written for a class when an ancestor is modified.

```
(defun eoops:load-class (nc c)
  (let ((np (map:get c 'parent)))
    (eoops:require-class np)
    (let* ((p  (map:get eoops:classes np))
           (pv (map:get p 'compiled))
           (cv (map:get c 'compiled))
           (s (map:get p 'children)))
      (map:set eoops:classes nc c)
      (map:set c 'modtime
               (eoops:class-mod-time nc 'obj))
      (map:set c 'compiled
               (eoops:inherit-behavior pv cv))
      (if p (map:set p 'children (cons nc s))))))
```

`eoops:write-class` writes the compiled class *c*, whose class name is *nc*, to the file *nc*`.elc` in the directory that is the value of `eoops:class-path`. The resulting file, when loaded into Emacs, will install class *c*. Each file contains only one class and each class has to be in exactly one file. However, a file may include other expressions. Therefore, `eoops:write-class` first byte-compiles the source file and then replaces the item corresponding to the class definition by the printed representation of the class. Setting `print-depth` to nil makes sure that the `prin1` prints the complete class. Since a class's children field is a list of its currently loaded subclasses, this field is set to nil before printing.

```
(defun eoops:write-class (c)
  (let* ((standard-output
          (get-buffer-create "*Compiled*"))
         (print-depth nil)
         (nc (map:get c 'name))
         (src-name (eoops:class-file-name nc 'src))
         (obj-name (eoops:class-file-name nc 'obj))
         (prefix
          (format "(eoops:load-class '%s '" nc))
         (suffix ")"))
    (message "Writing class %s..." nc)
    (byte-compile-file src-name)
    (set-buffer standard-output)
    (erase-buffer)
    (insert-file obj-name)
    (goto-char (point-max))
    (re-search-backward "^(class ")
    (delete-region (point)
                   (progn (forward-sexp) (point)))
    (insert prefix)
    (let ((s (map:get c 'children)))
      (map:set c 'children nil)
      (prin1 c)
      (map:set c 'children s)
      (insert suffix))
    (let ((make-backup-files nil))
      (write-file obj-name))
    (kill-buffer (current-buffer))
  (message "Writing class %s...done" nc)))
```

`eoops:class-file-name` returns the full path name of a file storing the class *nc*. *type* can either be `'src` or `'obj`.

```
(defun eoops:class-file-name (nc type)
  (format (cond ((eq type 'src) "%s/%s.el")
                ((eq type 'obj) "%s/%s.elc"))
          eoops:class-path nc))
```

`eoops:class-mod-time` returns the last modification time of the source or object file corresponding to the class *nc*.

```
(defun eoops:class-mod-time (nc type)
  (nth 5 (file-attributes
          (eoops:class-file-name nc type))))
```

`eoops:time-newer` compares two time values returned by `eoops:class-mod-time` and returns true if its first argument is greater than its second.

```
(defun eoops:time-newer (ta tb)
  (ce:let (((ta1 ta2) ta)
           ((tb1 tb2) tb))
    (or (> ta1 tb1)
        (and (= ta1 tb1)
             (> ta2 tb2)))))
```

4.9 Future Enhancements

This section describes future enhancements to the Eoops environment.

Earlier versions of Eoops had support for documentation and debugging which was lost with the removal of the Eoops interpreter. This could be reimplemented in the new compiled-only environment.

`eoops:class-path` should be a list of directories to search for class files.

Eoops could compile even better code by *inlining* method bodies. As this would eliminate all overhead for message-passing and function-calling, Eoops code could run faster than most hand-built Emacs-Lisp.

Message sends to known classes could be inlined. This includes all sends to `self` and `super`; sends to `self` typically comprise over 50% of the sends in an application. Eoops could allow the programmer to *declare* the types of slots and arguments; then the compiler could inline sends to them. Eoops could use type prediction or runtime compilation to inline *all* sends. All of the above compilation techniques are actually implemented in the SELF compiler (Chambers 91).

5 Experience using Eoops

We built Eoops in order to implement a remote evaluation facility for Emacs. This facility allows multiple Emacs processes to communicate by exchanging s-expressions. We then used the remote evaluation facility to implement several small interactive collaborative tools, including a mechanism for editing shared text in Emacs. We have built 45 object classes, representing about 2,600 lines of Eoops code.

6 How To Get It

You can FTP the latest copy of Eoops from `ftp.cs.ucla.edu` in the file `pub/eoops.tar.Z`.

7 References

(Borning 82a) Alan H Borning and Daniel H H Ingalls "A Type Declaration and Inference System for Smalltalk," *Ninth Symposium on Principals of Programming Languages,* 133–141, Albuquerque, NM, 1982.

(Borning 82b) Alan H Borning and Daniel H H Ingalls "Multiple Inheritance in Smalltalk-80," *Proceedings at the National Conference on Artificial Intelligence,* 234–237, Pittsburgh, PA, 1982.

(Chambers 91) Craig Chambers and David Ungar "Making Pure Object-Oriented Languages Practical," *OOPSLA '91,* 1–16.

(Goldberg 83) Adele Goldberg and David Robson *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading, MA, 1983.

(Ingalls 86) Daniel H H Ingalls "A Simple Technique for Handling Multiple Polymorphism," *OOPSLA '86,* 347–349.

(Lewis 90) Bil Lewis *et al. The GNU Emacs Lisp Reference Manual,* Free Software Foundation, Cambridge, MA, 1990.

(Lieberman 86) Henry Lieberman "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," *OOPSLA '86,* 214–223.

(McCall 80) Kim McCall "TinyTalk, a Subset of Smalltalk-76 for 64KB Microcomputers," *Sigsmall Newsletter,* September 1980.

(Moon 86) David A Moon "Object-Oriented Programming with Flavors," *OOPSLA '86,* 1–8.

(Snyder 86) Alan Snyder "Encapsulation and Inheritance in Object-Oriented Programming Languages," *OOPSLA '86,* 38–44.