Report 82-04
Stanford -- KSL

OMEN User's Manual.
Jay S. Lark,
Feb 1982

card 1 of 1

Stanford Heuristic Programming Project
Report No. HPP-82-4

February 1982

OMEN User's Manual

Jay S. Lark

Heuristic Programming Project
Department of Computer Science
Stanford University

This research was conducted under a Joint Study Agreement between International Business Machines and Stanford University, contract #2-FCZ903.

Table of Contents

1. Introduction

OMEN is an object-oriented programming system designed for use in a FRANZ LISP or other similar programming environment. OMEN stands for OBJECT MANIPULATION ENVIRONMENT, and consists of a set of functions to be loaded on top of a LISP system running MRS. The user can the use the functions provided by OMEN to create classes of objects, instances of those classes, and functions that operate on those objects, and to send messages to those objects.

OMEN is similar in design and operation to the *flavors* system of Lisp Machine Lisp and the LOOPS system for the Xerox Dolphin. OMEN was originally designed as a programming environment for an object-oriented graphics system, but the system should be useful for many different applications.

OMEN is not a programming language. It is a way of abstracting the data structures a program must use and the functions that operate on those data structures. Additionally, OMEN is not a mature system. It is constantly undergoing development to add new features and make the code more efficient. At any given time the actual system may not conform to this manual, but every attempt will be made to make the system upward compatible.

2. Objects and Classes

2.1. Objects

Objects are the most basic entity that OMEN deals with. An object can be any entity in a program. Typically, an object will have some *state* associated with it, such as a data structure. Examples of objects are buffers in an editting program or a device instantiation in a simulation program.

In OMEN objects are created by making an instance of a *class.* A class can be thought of as containing *prototypical* information for all members of that class. This prototypical information is in the form of *instance variables.* Instance variables are refered to by name, and can have as value any LISP object. When a class is instantiated to make an object all of the instance variables associated with that class are copied into the instance.[1]

As an example, consider the class of all ships. Suppose the state information of interest are the velocity, position, and mass of each ship. To create the class of ships with these instance variables type to OMEN:

```
     (DefClass ship (x-pos y-pos x-vel y-vel mass) nil)
```

Ignore the nil for the moment. Notice for this example the position and velocity have been split into their x and y components. To create an instance of a ship type to OMEN:

```
     (setq myship (MakeInstance nil 'ship))
```

Again ignore the nil for the moment. To describe the instance just created the Describe function can be used.

```
     (Describe myship)

     ship1 is an instance of the class ship with instance variables:

     x-pos  ==>  Unbound
     y-pos  ==>  Unbound
     x-vel  ==>  Unbound
     y-vel  ==>  Unbound
     mass   ==>  Unbound
```

2.2. Instance Variable Values

In the above examples the instance created had a complete set of instance variables, but these variables had no values. To be useful there must be a way of assigning values to the instance variables of an object. This can be done in several ways. One way is to provide a default value for instance variables when the class is created. For example:

---

[1]In the current implementation all state information *is* copied into a new instance. In new implementations it may be possible for objects to share one copy of certain instance variables.

```
(DefClass ship ((x-pos 0.0) (y-pos 0.0) mass
                 (x-vel 'Unknown) (y-vel 'Unknown))
                nil)
```

defines a class of ships which all have a default position of `(0.0 0.0)` and `Unknown` velocity.

Values can also can also be assigned to instances when they are created. This is done by adding variable/value pairs to the call to MakeInstance:

```
(setq mynewship (MakeInstance nil 'ship 'mass 200 'x-pos '3.0))

(Describe mynewship)

ship2 is an instance of the class ship with instance variables:

x-pos  ==>  3.0
y-pos  ==>  0.0
x-vel  ==>  Unknown
y-vel  ==>  Unknown
mass   ==>  200
```

In this case, the defaults from the class `ship` were used to provide defaults for the values of the instance variables, but the default for `x-pos` was overridden by the explict initialization value in the call to MakeInstance.

2.3. Class Hierarchy

In the previous section a method was shown for describing the class of ships and their instance variables. Suppose we also wanted to describe the class of automobiles, and include the same list of instance variables. This could be done with another call to DefClass, but this is not desirable. Using that approach, we would have to define the instance variables of every moving object with a similar call to DefClass. A better way is to create a class of moving objects, and then specialize that class to make more interesting classes.

Let the class `moving-object` be defined as `ship` was above. We can then define a new class of ships as follows:

```
(DefClass ship (name power) (moving-object))
```

The third argument to DefClass is a list of other classes that `ship` *inherits* state information from, in the form of instance variables. The list of classes is known as the *superclasses* of the class. If a superclass A itself has a superclass, say B, the instance variables of both A and B are inherited. The user can build up a *hierarchy* of classes, with the least specialized class at the root and the most specialized classes at the leaves. In fact, OMEN allows the user to define an arbitrary stucture of class relationships.

When OMEN instantiates a class to make an object, the superclass structure is searched in a depth-first manner to find instance variables and defaults to inherit. As an example, consider the classes defined below:

```
(DefClass foo (a (b 'init-b) c) (bar baz))

(DefClass bar ((a 'init-a1) (d 'init-d)) nil)

(DefClass baz ((a 'init-a2) e) nil)

(setq test (MakeInstance nil 'foo))

(Describe test)

foo1 is an instance of the class foo with instance variables:

a    ==>  init-a1
b    ==>  init-b
c    ==>  Unbound
d    ==>  init-d
e    ==>  Unbound
```

The initial value of a was determined by class `bar`, which was found first in searching the superclass tree. A class can only be searched once for instance variables. If there are multiple paths to a given class all but the first path to reach the class are ignored.

PAGES MISSING FROM ONLY EXTANT AND AVAILABLE COPY

3. Messages and Methods

So far this manual has shown how to define classes and instances of those classes in OMEN. This is still not adequate because there is no way to use the information stored in an objects instance variables. To solve this problem, objects in OMEN accepts *messages* and return *results*. These messages can request or set the value of an instance variable or carry out some arbitrary computation using the values of the instance variables. It is helpful to consider that all objects recognize a list of messages that it knows how to respond to.

3.1. Messages

When an object receives a message that it recognizes, it looks up the *method* that is uses to process that message and calls that method with the given arguments. The method is just a LISP function. As an example, the Describe function alluded to above actually send a `:print-self` message to the object. The object looks up the method for responding the `:print-self` message and calls it. At first, each object recognizes a small set of default messages. (See section 5 on Defaults.) A message is sent to an object with the function Send, which will be described further in section 3.2.

`(Describe foo)` is functionally equivalent to `(Send foo ':print-self)`

OMEN has the ability to automatically create a large number of messages and corresponding methods for accessing instance variables. Consider the example below:

```
(DefClass moving-object (x-pos y-pos x-vel y-vel mass) nil
        :settable-instance-vars
        (:gettable-instance-vars x-pos y-pos mass))
```

The keyword `:settable-instance-vars` means that a message and corresponding method should be created to set the value of each instance variable. For the `mass` variable the message will be `:set-mass`. The keyword `:gettable-instance-vars` creates a message for the named variables that returns that value of the variable. For example the name of the message to get the value of the `x-pos` variable will be `:x-pos`.

3.2. Definition of Send

Send is the user-level function used to send a massage to an object. When a Send is executed two operations take place. The first operation is to create a LISP environment in which all instance variables of the object are lambda bound to their respective values. The special variable `self` is bound to the name of the object. The method that responds to the given message is looked up. This method is then funcalled in the created environment with arguments `self` and the rest of the arguments to Send. Send returns the value of the method and restores the original environment.

An important point to note from the previous paragraph is that environment is created for the object. This allows methods to reference instance by name instead of using a separate fetch function. It is possible to send messages to an object already processing a message. To do this use the SendSelf function. This preserves the variable bindings and speeds the message sending process.

3.3. Inheritance of Messages

As mentioned previously, when an object is created in OMEN it inherits all the instance variables from all its superclasses. In an analogous way, an object also inherits all the messages recognized by all its superclasses. If the same message is recognized by more than one superclass, the superclass encountered first in the depth-first search is used, in exactly the same manner as instance variables.

3.4. Defining Methods

It is possible for a user to define methods for a class that will respond to a given message. Consider two examples relating to the class moving-object defined above:

```
(DefMethod :speed moving-object nil
           (sqrt (+ (* x-vel x-vel)
                    (* y-vel y-vel))))

(DefMethod :move moving-object (newx newy)
           (SendSelf ':set-x-pos newx)
           (SendSelf ':set-y-pos newy)
           (list newx newy))
```

In the first example, a method is defined for the class `moving-object` which responds to the `:speed` message. The method calculates the absolute value of the velocity. Notice that the instance variables are referred to by name.

In the second example, a method is defined for the class `moving-object` which responds to the `:move` message. In this case the method has two arguments `(newx newy)` which must be mentioned explicitly. The method also uses the SendSelf function. In this case, the `x-pos` and `y-pos` variables are set to `newx` and `newy`, respectively, and the objects returns `(list newx newy)` as its value.

A method can also use an already defined LISP function. In this case the function must be of one more argument than the number of variables in the lambda list. This is because the name of the object being processed is passed to any method's it uses as that methods first argument.

4. Functions, Options, and Variables

4.1. Functions

DefClass	`(DefClass name vars supers &rest options)` - *special form*
	Defines a class with the named superclasses. `vars` is a list of either variables or `(variable defaultvalue)` pairs. `defaultvalue` is not evaluated until an instance is created that uses that value. `options` can be a single keyword or a list whose car is a keyword. DefClass keyword options are listed in section 4.2.
MakeInstance	`(MakeInstance name class &rest init)` - *lambda*
	Creates a named instance of `class`. If `name` is `nil` then a new name is gensymed from the class name. The `init` arguments are `name value` pairs that correspond to instance variables of the object.
DefMethod	`(DefMethod message class lambdalist &rest forms)` - *special form*
	Creates a method for `class` that responds to `message`. If `lambdalist` is a non-nil atom then it is taken to be the name of a LISP function, which is then made the body of the method. If `lambdalist` is `nil` or a list then `forms` are taken to be the body of the method, and `lambdalist` is the list of lamda variables the method needs as arguments.
Send	`(Send object message &rest arguments)` - *macro*
	Sends `message` to `object`, and returns the result of the method invoked by that object.
↑	Shorthand form of Send.
SendSelf	`(SendSelf message &rest arguments)` - *macro*
	Sends `message` to the currently processing object. It is an error to use a SendSelf from the LISP top level.
↑↑	Shorthand form of SendSelf.
CompileMethods	`(CompileMethods &optional class file)` - *lambda*
	Writes some non-compiled methods to a file, compiles that file, and then loads back the object code. If `class` is given then only methods used by an instance of that class are compiled. If `class` is `nil` then all methods are compiled. If `file` is given then the source and object files that are created are moved to `file.l` and `file.o`, respectively.
Describe	`(Describe object)` - *lambda*

Prints the description of `object`. This is implemented by sending the `:print-self` message to `object`.

4.2. DefClass Options

:gettable-instance-vars
: Methods for returning the value of the named instance variable, or all instance variables if :gettable-instance-vars occurs alone, are created. The name of the messages are made by prefixing the name of the instance variable with the value of get-prefix.

:settable-instance-vars
: Methods for setting the value of the named instance variables, or all if :settable-instance-vars occurs alone, are created. The message name is formed by prefixing the name of the instance variable with the value of set-prefix. All methods created return the value that the variable was set to.

4.3. Variables

self	Self is always lambda bound to the name of the object currently processing a message. It is also passed as the first parameter to all methods.
get-prefix	Determines the prefix used in constructing get methods with the :gettable-instance-vars option of DefClass. Default is ":".
set-prefix	Determines the prefix used in constructing set methods with the :settable-instance-vars option of DefClass. Default is ":set-".

5. Default Messages

OMEN provides certain default message handling capabilities for objects. These methods are defined for the special class `class`. It is recommended that the most general class or classes of a user program use `class` as their superclass. `Class` accepts the following messages.

:init	Takes as arguments variable/value pairs and initializes the values of the named instance variables.
:print-self	Prints a description of the object.
:new-method	When OMEN creates a member of a class for the first time it sets up a fast lookup table for the messages and corresponding methods accepted by that class. If any new messages are created with DefMethod after the class has been instantiated, the fast lookup table will no longer be up to date. OMEN will run in this state with a performance degradation. To update the fast lookup table of a class send the `:new-method` message to any instance of that class. This is not necessary if the class has not yet been instantiated.
:which-operations	Prints a list of all messages that the object recognizes.
:get-method	Returns the name of the method for handling a given message. If the object does not recognize the message :undef-message is returned.
:eval-inside-yourself	Takes a single argument and evaluates it in an environment defined by lambda binding all instance variables of the object.
:funcall-inside-yourself	Takes a number of arguments and funcalls the first argument on the rest, in an environment defined as above.

6. Running OMEN

OMEN is available as a LISP core image on the diablo computer at Stanford University. To start OMEN type to the shell:

```
% /hpp/cmd/omen
```

OMEN first attempts to read the file `.omenrc`, first from the current directory and then the user's home directory. This leaves the user at the LISP top level.

Most error messages are self explanatory. If, when creating an instance, OMEN finds a reference to an instance variable it doesn't recognize it assumes a misspelling and ignores it. If an object is sent a message it doesn't recognize it ignores it and prints a warning message.

Any questions or comments regarding OMEN or this manual should be sent to lark@diablo.