PORK Object System Programmers' Guide

Ora Lassila

CMU-RI-TR-95-12

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

April 1995

This research has been supported in part by the Advanced Research Projects Agency under contract F30602-90-C-0119 "Flexible Multi-Agent Coordination" (as part of the ARPA/Rome Labs Planning Initiative) and under contract F30602-91-C-0014 (under subcontract to BBN Systems and Technologies), by the National Aeronautics and Space Administration (NASA) under contract NCC 2-531, and by the CMU Robotics Institute.

Abstract

PORK is an object system which brings a conventional object-oriented language closer to the requirements of frame-based programming. It only provides a very limited set of features (on top of the base object system itself), and aims to achieve seamless integration with conventional programming. PORK is implemented as a portable metalevel extension of the Common Lisp Object System (CLOS). It extends CLOS by adding the following concepts:

- Named objects. These simplify debugging and linked frame model construction. Collections of named objects (called "namespaces") can also be used as a low-level implementation vehicle for knowledge bases.
- References to nonexisting named objects. Programs can manipulate objects that have not yet been created. This greatly simplifies definition of complex linked frame models with circular references.
- Slots with multiple values, and a mechanism for defining an access interface.
- Automatic updating of inverse slots.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Overview of Features 3

2 Programming with PORK **7**

2.1 Mixing PORK and "Plain" CLOS 7

2.2 Using Many-Valued Slots 8

2.3 Using Inverse Relations 9

2.4 Defining Your Own Access Methods 11

2.5 Implementing Daemons 15

2.6 Using Slot Options 18

2.7 Extending the System 20

2.8 Using Namespaces 21

3 Programming Interface **25**

3.1 Named Objects 25

3.2 Class Definition 28

3.3 Instance Creation 29

4 Metaprogramming Interface 31

4.1 Named Objects 31

4.2 Namespaces 33

4.3 Forward References 36

4.4 Frame Objects 37

4.5 Frame Classes 38

4.6 Relations 41

5 Design and Implementation Notes 45

5.1 Object Naming Mechanism 45

5.2 Forward References and Deferred Access 46

5.3 Class Definition 48

5.4 Instance Creation and Initialization 49

5.5 Common Lisp System Compatibility 50

5.6 Reliance on Common Lisp MOP 51

5.7 Porting to Other Platforms 53

Acknowledgements

The PORK system, officially started in 1992, was designed for use in the re-engineering of the OPIS/DITOPS scheduling system. The work was supported in part by the Advanced Research Projects Agency under contract F30602-90-C-0119 "Flexible Multi-Agent Coordination" (as part of the ARPA/Rome Labs Planning Initiative) and under contract F30602-91-C-0014 (under subcontract to BBN Systems and Technologies), by the National Aeronautics and Space Administration (NASA) under contract NCC 2-531, and by the CMU Robotics Institute.

The questions and comments from the users of PORK often made me notice confusing details either in the system or in the early drafts of the manual (this is not to say that there wouldn't be any confusing things in this document). I would like to thank all the people who in this way (or various other ways) contributed to the design of PORK: Gilad Amiri, Marcel Becker, Dina Berkowitz, Ali Safavi, Stephen F. Smith and Chris Young. I owe special gratitude to Steve for the faith he had in letting me take a drastic approach to the redesign of DITOPS.

Since PORK is the long-awaited *Well Done BEEF*, I would also like to extend my thanks to all the people at Helsinki University of Technology who helped me during the initial design phase of WDB. Last, but most certainly not least, for all her encouragement I would like to dedicate this work to my wife Marcia who by now is probably sick and tired of all these meat products...

Pittsburgh, April 1995

Ora Lassila

Chapter 1

Introduction

Initially, PORK was designed for use in the re-engineering of a large knowledge-based system, the OPIS/DITOPS scheduling system [13, 14, 15, 16]. The purpose of the re-engineering effort was to move from an old frame-based implementation to a more maintainable object-oriented version. This chapter introduces PORK and provides some background for the motivation of its design.

The name PORK stands for "Programmable Objects for Representing Knowledge." (earlier the system was called "Parsifal" but because of a conflict with the name of an existing system this name had to be dropped). From the design standpoint PORK has its heritage in the "well done BEEF" efforts [11] undertaken at the end of the BEEF project [3, 9, 10].

1.1 Motivation

The concept of frames [12] and the idea of a frame system have been very popular as an easy-to-understand paradigm for representing complex data. The fundamental idea is simple: *frames* are the system's basic objects, they represent real-world concepts and phenomena; frames have named attributes, *slots*, and slots can be assigned *values*. Inheritance allows slot values to be used as defaults. From the standpoint of programming it is perhaps unfortunate that what frame systems are really good for is *representing* things. In an attempt to make it easier to use frame systems in a normal programming context, designers of frame systems have often added features that add "object-oriented" programming capabilities to frame systems (examples of systems like this are KEE [1, 18], CRL [19] and BEEF [3, 9, 10]).

There are differences between "real" object-oriented programming and the frame-based flavor of object programming. First, there are definite differences between inheritance mechanisms: In object-oriented programming systems, types usually form a taxonomy, and inheritance is understood as the propagation of functional and structural information in this hierarchy. Often the propagation takes place at definition time; when an object-oriented program is running the inheritance hierarchy is more or less fixed. In frame systems, however, inheritance is used to propagate default slot values in the type hierarchy. Inheritance is typically dynamic in nature, i.e. takes place at run time. This allows default values to be changed during program execution. Frame systems may also allow the use of alternate inheritance paths in addition to the usual conceptual subsumption. These differences in inheritance are closely related to differences in program access to the type hierarchy. In an object-oriented programming system, only instances are objects of the system. Systems that offer facilities for *metaprogramming* [2, 17] also allow the type hierarchy to be accessed. In some frame systems types do not fundamentally differ from instances, and program access to the type hierarchy is possible without any meta level.

What frame systems lack with respect to object-oriented programming systems is proper facilities for data hiding and support of encapsulation. Object-oriented programming systems typically support and enforce encapsulation, allowing objects to be manipulated through a well-defined function interface. Frame systems do not usually restrict access to slot values of any frames in any way.

Frame systems, as opposed to object-oriented *programming* systems, are usually designed for *representing knowledge*, with the programming aspect not necessarily considered very important. Object-oriented programming systems in general are not adequate for knowledge-based systems programming. For a more detailed discussion on frame systems vs. object-oriented programming systems, see for example [8]. Many attempts have been made to bring frame-based programming closer to mainstream programming by adding object-oriented programming features. On the other hand, PORK brings some of the features of frame systems into the realm of mainstream object-oriented programming. PORK achieves this by adding some useful features from frame systems into a standard object-oriented programming language.

1.2 Overview of Features

PORK is an object system which brings a conventional object-oriented language closer to the requirements of frame-based programming. It only provides a very limited set of features (on top of the base object system itself), and aims to achieve seamless integration with conventional programming. The design of PORK draws from experience with frame systems like CRL and KEE, and from the insight into frame system design gained during the BEEF project [10].

PORK is implemented as a portable extension of the Common Lisp Object System (CLOS, [17, pp.770-864]). It extends CLOS by adding the following concepts:

- Named objects.
- References to nonexisting (future) named objects.
- Slots with multiple values, and a mechanism for defining an access interface to these.
- Automatic updating of inverse slots (these are similar to CRL's inverse slots).

Otherwise PORK behaves exactly like standard CLOS, thus allowing PORK and CLOS code to be freely mixed. PORK also has a *metaobject protocol* to allow the system to be extended (for an introduction of metaobject programming we recommend the book "The Art of the Metaobject Protocol" [6] which explains the philosophy of metaobject protocols as well as documents the proposed standard MOP for CLOS).

PORK, as an extension of CLOS, adheres to the *class-instance* paradigm. Other frame systems, where the distinction between class-instance and *prototype-instance* approaches is blurred (or those which completely adhere to the prototype-instance paradigm), allow the programmer to do things which are not possible in PORK. The most notable omissions are:

- Dynamic addition of slots: In PORK, all slots have to be defined in the class. Furthermore, the whole idea of slot is different since a strict access protocol is used; something that "looks like" a slot may not be one.

- Dynamic inheritance: Changing slot default values specified in a class is a "meta-level" operation and not necessarily meant to be done during program execution. There is no mechanism for changing a default value and propagating this change to existing instances which have not modified the slot in question.

1.2.1 Named Objects

An ordinary CLOS instance does not have a name. Reference to an instance is possible only if a "link" exists from your data to the particular instance (either the instance is a member of your data structures, or it is stored in a variable). The PORK naming mechanism provides a mapping from names to instances. This mapping is beneficial for the following reasons:

- It simplifies instance references. One only has to know a name, and does not have to store the instance in a specific place just because one would later want to access it. This is very convenient when debugging.
- It simplifies instance identification. Symbolic names are easier to remember and identify than machine addresses in hexadecimal format (and these may even change during the execution of a program). Again, this is a feature that makes debugging programs easier.
- It makes it possible to refer to instances without the evaluation process. In other words, it is possible to write constants which have instance references. This simplifies writing complicated linked instance models.

PORK prints named instances in a different manner than ordinary (anonymous) instances. Here is an anonymous instance (from an imaginary class `person`):

```
#<person 0xEF9834>
```

whereas here is a named instance, called `john-doe`:

```
#<person john-doe>
```

PORK has a definition mechanism for the declarative creation of instances; the definition macro is called `definstance`. Here is an example of how the instance called `john-doe` might be created:

```
(definstance john-doe (person)
  :full-name "Doe, John")
```

Here it has been assumed that the `person` class has an initialization parameter called `:full-name` (one should not get confused, this initialization parameter is here only for the sake of the example, it has nothing to do with the instance naming mechanism). When creating instances using `make-instance`, the instance name is given using the initialization parameter `:name`.

PORK also extends Common Lisp syntax by allowing a shorthand notation for referring to named objects. To create a reference to an instance called `john-doe`, one needs to write `!john-doe`. This notation is resolved at the time the expression is read in. Please note that `!john-doe` is not the name of the instance, it is *the instance itself*. Symbols that name objects are different from the objects themselves (this is different from the approach taken, for example, in CRL and BEEF, where the symbols that name objects *are* the actual objects).

If a reference is made to an instance that does not exist (i.e. the name is not found in PORK's internal symbol table), a *forward reference* is created. A forward reference is an object that represents the actual (future) instance until it gets created. Of course, it is not completely equivalent to the instance it represents (because, for one, it is of different class), but it will serve as an object that you can pass on to other objects, store in data structures etc. When an attempt is made to create the actual instance, the forward reference is *changed* using the CLOS function `change-class` and it becomes the new instance. That way, references made before the creation are valid and need not be updated.

One can also call slot accessors on forward references (more specifically, slot writers, adders and removers; these are introduced in the next section). Especially in the case of automatic inverse relations (again, refer to a subsequent section describing these) it may happen that a slot writer gets called on an object that "does not exist" (i.e., a forward reference). These calls are deferred and executed when the real object gets created. That way one can easily create cyclical structures (e.g., doubly linked data structure).

The symbol table used by PORK can be changed, and the user can create new symbol tables. In PORK, these symbol tables are called *namespaces*.

1.2.2 Relations

Relations in PORK consist of all slots with special behavior. These include *many-valued* slots, *inverse single-valued* slots and *inverse many-valued* slots.

PORK implements many-valued slots as CLOS slots where the values are stored as a list (these slots used to be called "multiple-valued" slots, but this name is misleading because of the Common Lisp connotation of the term "multiple values"). To make sure that nothing undesirable happens to the list of values, a strict access protocol is enforced. This is done through the use of three new types of slot accessors (compare these to the ones CLOS normally has: *readers* and *writers*):

- *Adders* are accessor functions that add values to a slot. Calling an adder once adds one value to a slot.
- *Removers* are accessor functions that remove values from a slot. Calling a remover once removes one value from a slot.
- *Cleaners* are accessor functions that remove *all* values from a slot.

Here is an example of the use of an adder function (this assumes that `!john-doe` and `!jane-doe` are instances of the class `person` which has an adder function called `add-child`):

```
(add-child !jane-doe !john-doe)
```

i.e. `!jane-doe` was added to the list of children of `!john-doe`. Note that the value to be added *precedes* the object to be added to. This is different from, say, the convention of CRL where `add-value`'s first argument is the object to be added to. The reverse order has been chosen so that adders and removers would be similar to CLOS writer methods.

If you define a slot with an adder and/or a remover accessor, you cannot use the CLOS slot options `:writer` or `:accessor`, since the existence of these functions would allow you to violate the many-valued slot access protocol. A cleaner can only be defined if the slot has an adder and a remover.

Chapter 2

Programming with PORK

This chapter will give a "hands-on" introduction to programming techniques in PORK. The examples will build gradually, introducing more features and techniques step by step; the examples are drawn from the production scheduling domain, modeling manufacturing facilities.

This manual is not an introductory text to programming with CLOS. Readers unfamiliar with CLOS programming techniques are first encouraged to study some CLOS programming textbook, for example [5].

2.1 Mixing PORK and "Plain" CLOS

Since PORK is just an extension of CLOS, mixing PORK and CLOS programming is possible without trouble. For example, one can define ordinary slots to frame classes, and these will behave exactly like they would in "plain" CLOS. For example, to model manufacturing resources we can define a frame class called `resource`. We will later introduce more details to this class:

```
(defframe resource (frame-object)
  ((total-capacity
     :initarg :total-capacity
     :reader total-capacity)))
```

Please note that in PORK we use the definition form `defframe` instead of the usual `defclass`; the parameter syntax is the same. Furthermore, objects defined using `defframe` have to inherit from `frame-object` (this is analogous to CLOS classes inheriting from `standard-object`, except that in PORK the insertion of the base class into the class precedence list has not been automated). One can also make frame classes inherit from ordinary CLOS classes, but one is not allowed to redefine their existing slots to be many-valued slots. Any new slots that one defines can use many values, however.

2.2 Using Many-Valued Slots

Access to many-valued slots is effected through *adders*, *removers* and *cleaners*. Here is an example of how to define an adder and a remover:

```
(defframe resource-group (resource)
  ((members
     :initarg :members
     :initform nil
     :reader members
     :adder add-member
     :remover remove-member)))
```

The class `resource-group` gets created, together with the accessor functions `members` (for reading the values of the slot), `add-member` (for adding values to the slot) and finally `remove-member` (for removing values from the slot). The values of the slot can be initialized when an instance of `resource-group` is created, by specifying the `:members` initialization parameter. Its value has to be a list, of course; elements of that list will get added to the slot, one at a time, using the adder function. Here is an example:

```
(definstance drilling-cell-1 (resource-group)
  :members '(!drill-1 !drill-2 !drill-3))
```

Since the values of a slot are kept in a list, and this list is returned as a result of a call to the slot's reader, it is an error to surgically modify this list (i.e. to use functions like `sort`, `delete`, `nconc` etc.). If one needs to modify a list returned by a many-valued slot's reader, the function `copy-list` should be used first.

2.3 Using Inverse Relations

Let us assume a definitions of classes called `resource` and `resource-group`. Now, `resource-group` maintains a list of its members. To make individual resources aware of which groups they belong to, we will define a slot `groups` in the `resource` class to be the *inverse* of the slot `members` in the class `resource-group`. When either slot gets modified, the other is *automatically* changed, too.

```
(defframe resource (frame-object)
  ((total-capacity
     :initarg :total-capacity
     :reader total-capacity)
   (groups
     :initform nil
     :reader groups
     :adder add-group
     :remover remove-group
     :inverse (:adder add-member
               :remover remove-member))))

(defframe resource-group (resource)
  ((members
     :initarg :members
     :initform nil
     :reader members
     :adder add-member
     :remover remove-member
     :inverse (:adder add-group
               :remover remove-group))))
```

Now the functions `add-member` and `add-group` are each others' inverse adders, and correspondingly `remove-member` and `remove-group` are each others' inverse removers. Notice how the definition of `resource-group` was changed to reflect the introduction of the inverse relation.

If one wants to use single-valued slots (i.e. ordinary CLOS slots) with inverse updating, it is possible by specifying an accessor and an inverse accessor (instead of adders and removers). For example, if we have an object class to represent a resource time line (e.g., planned schedule for a resource), say, `time-line`, then the resource and the time line can be linked using single-valued inverse relations:

```
(defframe resource (frame-object)
  ((time-line
     :initarg :time-line
     :accessor time-line
     :inverse (:accessor resource))
   ...))

(defframe time-line (frame-object)
  ((resource
     :initarg :resource
     :initform nil
     :accessor resource
     :inverse (:accessor time-line))))
```

Executing something like

```
(make-instance 'time-line :resource !drill-1)
```

would make `!drill-1` the resource of the created time line, and automatically the time line of `!drill-1` would be the created time line. One can also make the accessor and the inverse accessor the *same function*, though it is probably more typical for them to be different functions. An example of a function being its own inverse accessor would be in an example of a person and his/her spouse:

```
(defframe person (frame-object)
  ((spouse
     :initarg :spouse
     :initform nil
     :accessor spouse
     :inverse (:accessor spouse))))
```

2.4 Defining Your Own Access Methods

The protocol used by adder and remover methods is documented. It is quite possible to define your own methods that behave the same way as system-generated access methods.

2.4.1 Defining Adders and Removers

Sometimes adding an object to a many-valued slot is a more complicated operation than what the default adder methods assume. For example, if we want to maintain lists of operations executed by the resources we are modeling, instead of defining just one many-valued slot we may want to categorize operations into those currently in process and those still waiting to be executed. The class definition would thus look like this:

```
(defframe resource (frame-object)
  ((waiting-operations
     :initform nil
     :reader waiting-operation
     :adder add-waiting-operation
     :remover remove-waiting-operation)
   (in-process-operations
     :initform nil
     :reader in-process-operations
     :adder add-in-process-operation
     :remover remove-in-process-operation)
   ...))
```

This would give us two adders. Now, if we want to define a single adder method for adding an operation (say, `add-operation`) and make that decide which lower-level adder to use, we can do it like this:

```
(defmethod add-operation (operation (resource resource)
                          &optional updatep)
  (declare (ignore updatep))
  (if (in-process-p operation)
```

```
    (add-in-process-operation operation resource)
    (add-waiting-operation operation resource)))
```

Notice the optional parameter *updatep* which we just ignore. If one wants to define a method compatible with the inverse slot mechanism this parameter becomes significant, because it used to avoid infinite cyclical updating of inverse slots. Any client call to an adder method passes `nil` for *updatep* (that is, does not pass the parameter in the call). The adder method calls the inverse adder with the corresponding *updatep* as `t`, signaling that the other adder method is not supposed to make a call back to the first adder.

To complete the example, let us assume that a class called `operation` has been defined, and that it has an adder called `add-resource` with a specified inverse adder called `add-operation`. This is how our own definition of `add-operation` should be written:

```
(defmethod add-operation (operation (resource resource)
                          &optional updatep)
  (if (in-process-p operation)
    (add-in-process-operation operation resource)
    (add-waiting-operation operation resource))
  (unless updatep
    (add-resource resource operation))
  operation)
```

By convention, adders and removers always return the added or removed object. The remover method for this example is analogous to the adder.

2.4.2 Implementing 1-to-many Relations

Note that all relations in PORK are either 1-to-1 or many-to-many relations. To implement a 1-to-many relation, many-valued slots should be used and an additional reader defined which always returns the first value of the slot in question.

```
(defframe operation (frame-object)
  ((resources
```

```
      :initform nil
      :reader resources
      :adder add-resource
      :remover remove-resource
      :inverse (:adder add-operation
                :remover remove-operation))))

(defmethod initialize-instance :after ((self operation)
                                       &key resource)
  (when resource
    (add-resource resource self)))

(defmethod resource ((self operation))
  (first (resources self)))
```

This way the `operation` class has a reader method `resource` that always returns a single resource object, and an initialization parameter `:resource` for initializing an operation instance with a single resource. If one "hides" the definitions of the original reader, adder and remover methods (e.g., by not exporting them) this relation becomes a 1-to-many "initialize and read only" relation. To add a writer method one would have to define a cleaner (say, `remove-all-resources`) to the class definition and a method definition as follows:

```
(defmethod (setf resource) (resource (self operation))
  (remove-all-resources self)
  (add-resource resource self))
```

A future version of PORK may implement automatic generation of 1-to-many relation access functions.

2.4.3 Implementing Asymmetric Inverse Relations

Sometimes you have a situation where you want one particular access function to be the inverse of two different access functions. An example of this would be a tree structure where you want to maintain upward links. A problem is caused by the fact that although the inverse for each of the *downward* links is unambiguous, this is not the case for *upward* links.

As an example, let us consider a binary tree: we introduce slots named `left-child`, `right-child` and `parent`. This situation is problematic from the PORK standpoint because of two reasons:

- The system should not define the automatic inverse relation for `parent`, because its semantics are ambiguous (should it update the left or the right child). In fact, there is never a reason for a user program to set the parent slot.
- One cannot define the inverse for `left-child` and `right-child` and then not define the inverse for `parent`, because if the parent slot has no PORK-specific features, it becomes an ordinary CLOS slot, and its `setf` method will have a different signature than what the system expects of inverse accessors (PORK inverse accessors accept one optional argument).

To define the accessors correctly you have to do as follows: First, define the slots `left-child` and `right-child` by specifying that their inverse accessor is the accessor for `parent`, and second, do not declare an accessor for `parent`, but define the method yourself. Here is the implementation:

```
(defframe binary-tree-node (frame-object)
  ((left-child
     :initarg :left-child
     :initform nil
     :accessor left-child
     :inverse (:accessor parent))
   (right-child
     :initarg :right-child
     :initform nil
     :accessor right-child
     :inverse (:accessor parent))
   (parent
     :initform nil
     :reader parent)))

(defmethod (setf parent) (parent (child binary-tree-node)
                          &optional updatep)
  (assert updatep)
  (setf (slot-value child 'parent) parent))
```

The important thing is that the `setf` of `parent` is never called explicitly, only the automatic inverse update mechanism calls it. You might even want to name the `setf`-method differently in order not having to export it. The `assert` form has been placed in the function as a "safety check" that all calls actually are "update" calls from other accessors.

2.5 Implementing Daemons

PORK offers many possibilities for more advanced techniques than what have been described so far. One should bear in mind that PORK is a CLOS program *and* an extension of CLOS, thus all the techniques available for CLOS programmers are available when programming with PORK.

In PORK, all slot accesses are assumed to be handled by the accessor functions. Direct slot access is not allowed with PORK programs, since it would easily violate the slot access protocol (technically it is still possible with `slot-value`). PORK's accessor methods are defined *on top of* `slot-value`. Since a well-defined access interface can be relied on, defining additional slot value manipulation using auxiliary (i.e., `:after`, `:before` and `:around`) access methods is possible and reliable. This offers possibilities for defining "daemonic" behavior, i.e. actions triggered as a result of values being added to a slot or being removed from a slot.

Our example of resources and resource groups can be extended by defining the total capacity of a resource group to be the sum of the capacities of its members. We can implement this in several ways:

- Dynamic recalculation of the summed capacity is the easiest (and probably the least efficient way):

  ```
  (defmethod total-capacity ((self resource-group))
    (loop for member in (members self)
          sum (total-capacity member)))
  ```

  Readers not familiar with the Common Lisp `loop` macro are encouraged to see [17, pp.709-747, especially p.733].

- Incremental modification of the total capacity "caches" a previously computed capacity value for immediate access (it is a reasonable assumption that reading the capacity value is a more frequent operation than adding members to or removing members from a group). The class definition looks like this:

```
(defframe resource-group (resource)
  ((total-capacity
     :initform 0
     :writer (setf total-capacity))
   (members
     :initarg :members
     :initform nil
     :reader members
     :adder add-member
     :remover remove-member
     :inverse (:adder add-group
               :remover remove-group))))
```

The "daemons" look like this:

```
(defmethod add-member :after (resource
                              (group resource-group)
                              &optional updatep)
  (declare (ignore updatep))
  (incf (total-capacity group)
        (total-capacity resource)))

(defmethod remove-member :after (resource
                                 (group resource-group)
                                 &optional updatep)
  (declare (ignore updatep))
  (decf (total-capacity group)
        (total-capacity resource)))
```

This way, every time a member is added to the group the total capacity of the group is increased, and when a member is removed the capacity is decreased.

- Yet another way of implementing this is to use a combination of the above two techniques. The first time the total capacity is accessed it is computed and cached, after that incremental changes are made. The example would now read:

```
(defframe resource-group (resource)
  ((total-capacity
     :initform nil
     :writer (setf total-capacity))
   (members
     :initarg :members
     :initform nil
     :reader members
     :adder add-member
     :remover remove-member
     :inverse (:adder add-group
               :remover remove-group))))

(defmethod total-capacity ((group resource-group))
  (or (call-next-method)
      (setf (total-capacity group)
            (loop for member in (members self)
                  sum (total-capacity member)))))

(defmethod add-member :after (resource
                              (group resource-group)
                              &optional updatep)
  (declare (ignore updatep))
  (when (slot-value group 'total-capacity)
    (incf (total-capacity group)
          (total-capacity resource))))

(defmethod remove-member :after (resource
                                 (group resource-group)
                                 &optional updatep)
  (declare (ignore updatep))
  (when (slot-value group 'total-capacity)
    (decf (total-capacity group)
          (total-capacity resource))))
```

There are a couple of things that we have to point out: (1) it is very important for the initial value of the `total-capacity` slot to be `nil` (this ensures the condition for triggering the initial computation of the value), and (2) to use `slot-value` and not the defined reader method to access the slot in the "daemon" methods (this prevents a premature initial computation of the value).

> The net effect of this approach is that members can be added or removed but they will not affect the total capacity until the total capacity is accessed for the first time. It is left as an exercise to the interested reader to implement the case where the capacity of a member resource can change during execution.

In a simple example like this one there is probably no need to try to optimize the performance using these techniques, but it is easy to imagine more complicated situations where caching and deferred computation can have significant performance ramifications.

2.6 Using Slot Options

PORK introduces some new slot options in addition to those of standard CLOS. The options for defining new access methods have already been discussed in previous sections. The remaining options deal with how slot values are maintained and how they are saved into binary files.

2.6.1 Saving Slot Values

Sometimes one creates instances that are saved into a binary file as a result of compilation. For this purpose CLOS has a generic function called `make-load-form` which is called by the compiler to produce a representation for an object which can be compiled and stored into a binary file. This representation consists of Common Lisp forms which, when executed, produce an equivalent object and initialize it accordingly.

Because of many-valued relations and instance naming, PORK instances have to be saved in a special way. For this purpose, the class `frame-object` has a method for `make-load-form`. If `*save-named-objects-as-forward-references*` is true (the default), `make-load-form` will not be called at all.

In case of inverse relations, we typically want to save only one half of the relation (to use a previous example, only `members` would be saved, and `groups` would be created automatically at load time). It is therefore possible to specify whether `make-load-form` should save or discard the values of a slot: this is done using the `:save-values` slot option:

```
(defframe resource (frame-object)
  ((total-capacity
     :initarg :total-capacity
     :reader total-capacity)
   (groups
     :initform nil
     :reader groups
     :adder add-group
     :remover remove-group
     :inverse (:adder add-member
               :remover remove-member)
     :save-values nil)))
```

The default value for this option is `t`.

2.6.2 Slot Value Maintenance

By default, many-valued slots are implemented as CLOS slots with a list as their value. The list holds the multiple values of the PORK relation. The maintenance of this list is done using *copying* list operations (more specifically, removal of a value is done by substituting the list with a copy where the deleted value is missing). This way any lists of values returned by a many-valued slot reader are not affected.

Sometimes this approach can result in excessive "trashing" (imagine a large set of values, and removing these values starting from the end). For those situations, the `:allocation` slot option can be specified as `:fragile`. This instructs the system to use *destructive* list operations when removing values. The flip side of this is that any sets of values returned by a many-valued slot reader would get modified as well. Use the `:fragile` option with extreme care.

It should be mentioned that class allocation of relation slots is not allowed in PORK (specifying the value `:class` for `:allocation`). Ordinary CLOS slots can use class allocation.

2.7 Extending the System

Terminology note: According to [6], metaobjects (i.e., classes, generic functions, methods etc.) can be divided into three categories: those defined in CLOS MOP are called *specified*, those defined by an implementation (of CLOS) are called *implementation-specific*, and those defined by a portable program are called *portable*. In this document, we use the word *specified* to mean those metaobjects that are either defined in CLOS MOP or in the PORK metaprogramming interface (chapter 4). In practice, specified and implementation-specific entities are those residing either in one of the system packages or in the `pork` package.

Because PORK is a CLOS program and it has its own extensions to the CLOS metaobject protocol, it is possible to extend the functionality of PORK. Sometimes extending requires the programmer to write her own subclasses to specified classes or to define her own methods to specified generic functions. In order to avoid clashes with specified methods, one should observe the following rules (adapted from [7, 6]):

- Redefinition of specified metaobjects (classes, generic functions, etc.) is forbidden.
- User-defined metaobjects must be named in a user-defined package.
- When defining a method for a specified generic function, at least one of the specializers must be a user-defined class or an `eql`-specializer whose associated value is not an instance of a specified class. This way it is impossible for the method to "clash" with an existing method.

The PORK system consists of several components which can be used and extended independently of each other. It is important to understand that when defining classes that inherit from the base class `frame-object`, one has to use the definition macro `defframe`; otherwise the standard definition macro `defclass` should be used (among other things `defframe` will ensure that the metaclass `frame-object` – or one of its subclasses – is used).

2.7.1 Initialization of Instances

New methods specialized for frame classes for any of the standard generic functions may be defined, but one must not disable the inherited initialization mechanism.

In practice this means that if a primary method or an `:around` method for the following generic functions is defined, the inherited method *must* be called (using `call-next-method`):

- `initialize-instance`
- `reinitialize-instance`
- `shared-initialize`
- `make-instance`

For further details on instance initialization, see [17, pp.801-810]. The object naming mechanism relies heavily on the `make-instance` method specialized for the metaclass `frame-class`; it is therefore always necessary to call this method (even if `make-instance` was specialized for a new metaclass).

2.8 Using Namespaces

Namespaces are primarily intended to provide the object naming mechanism with future object references. They can also be used independently to provide other types of services.

One of the key services of namespaces is to store forward references that are later to be changed to "real" instances. The function `namespace-unresolved-forward-references` can be called to retrieve a list of all (currently) unresolved forward references. The current implementation loops through all the objects in the namespace and returns the forward references. If the performance of this function is critical, it can be speeded up by caching forward references as they are created. For example:

```
(defframe fw-caching-namespace (namespace frame-object)
  ((forward-references
     :initform nil
     :reader namespace-unresolved-forward-references
     :adder namespace-add-forward-reference
     :remover namespace-remove-forward-reference)))
```

```
(defmethod (setf namespace-find-named-object) :after
           ((object forward-reference) name
            (self fw-caching-namespace))
  (namespace-add-forward-reference object self))
```

Note that the form `defframe` was used here because we wanted to use many-valued slots. Otherwise a simple `defclass` would have done.

A mechanism will also have to be designed that removes objects as they are converted from forward references to real frame instances. This is one way of doing that:

```
(defclass cached-fw (forward-reference)
  ())

(defmethod update-instance-for-different-class :after
           ((old cached-fw) (new frame-object) &key)
  (namespace-remove-forward-reference old *namespace*))
```

To automatically create correct forward references, this method definition is also required:

```
(defmethod namespace-forward-reference-class
           ((self fw-caching-namespace))
  'cached-fw)
```

In this example a new forward reference class was defined, and a method which will remove the forward reference from the namespace when it is converted into a proper frame instance. Instead of using `*namespace*` a safer way is to use `object-namespace`. This function, however, is not necessarily optimized for this purpose, so the new forward reference class may want to store its namespace in a slot in the object itself.

2.8.1 Using a Persistent Store

You may want to use namespaces as an interface – and a caching mechanism – to a persistent object store. We are not suggesting implementing object databases using namespaces, but often one encounters situations where objects or other information needs to be retrieved by name or using other criteria from a secondary storage medium, and one really doesn't want to keep all of this information in the Lisp address space all the time.

When subclassing `namespace` for this purpose, there are several important generic functions to take into account. They will be discussed in this section.

The interface to retrieving and inserting objects into the database are the functions `namespace-find-named-object` and its `setf`-method. For example, if one wants the new namespace class to just be an interface to a database retrieval mechanism, this is how it can be done (we assume the existence interface functions `db-get` and `db-put` for reading and writing the database):

```
(defclass db (namespace)
  ())

(defmethod namespace-find-named-object (name (self db))
  (db-get (string name)))

(defmethod (setf namespace-find-named-object) ((object named)
                                                name
                                                (self db))
  (db-put (string name) object))
```

The method `namespace-find-named-objects` can be overridden to provide an interface to the object store's query mechanism.

This example represents an uncached interface. If one wants to use the namespace as an intermediary cache between Lisp and the object store, then probably a `:before` method for `namespace-forget-named-objects` is required for "flushing" modified objects back to the store. For example, the method can call a function like this:

```
(defun flush-modified-objects (db)
  (dolist (object (namespace-find-named-objects db
                    :predicate #'modifiedp))
    (db-put (string (object-name object)) object)))
```

Deferred database access can be implemented using forward references. The namespace can create forward references for unknown objects, and they can (at some suitable point during the program execution) be collected using the function `namespace-unresolved-forward-references` and perhaps retrieved from a database using an optimized query.

Chapter 3

Programming Interface

This chapter contains descriptions of all the functions, macros and variables needed to program with PORK. Since PORK just extends CLOS functionality, this chapter is very short. Many of the "interesting" details of the system are described in the chapter describing metaprogramming (chapter 4).

3.1 Named Objects

The instance naming mechanism is based on the class `named` which described in the metaprogramming section of this manual. In general, named objects are stored in a symbol table (they are called *namespaces*), and the function `find-named-object` allows one to retrieve existing objects from the symbol table.

A special readtable has been created, allowing one to refer to named objects using shorthand notation. The exclamation point (`!`) is used as the macro character, so named object references look like this: `!foo`.

When a reference to a named object is made so that the particular object has not yet been created, a *forward reference* is created. These forward references are instances of the class `forward-reference` (also described in the metaprogramming manual).

Normally, all named objects are printed using the standard Common Lisp unreadable object printing convention. If the standard special variable `*print-readably*` is true, named objects are printed using their exclamation-point syntax.

`object-name` [Generic function]
thing

This function will return the name of a named object. For objects not inheriting from any named object class, the default method returns the object itself.

`object-name` [Method]
(*self* `t`)

This method exists so that all lisp objects would have a method for object name. This methods returns *self*.

`*namespace*` [Variable]

This variable holds the current namespace object. Initially it is bound to the value of the system's *root namespace*. All named object operations involving a namespace use the value of this variable.

To change the current namespace this variable can be bound. It should not be assigned directly.

`find-namespace` [Function]
name

This function performs the mapping between namespace names and actual namespace objects. If a namespace for a given name does not exist, `nil` is returned.

The system's root namespace has the name `nil`.

`find-named-object` [Function]
name

This function returns the object named using *name*. If such an object does not exist, `nil` is returned. The object lookup is done by calling `namespace-find-object` on the current namespace.

`ensure-named-object` [Function]
name
`&optional` *error-if-does-not-exist-p*

This function returns a named object called *name*. If such an object does not exist, the parameter *error-if-does-not-exist-p* controls the action taken: if false, a forward reference is created; if true, an error is signalled.

`*save-named-objects-as-forward-references*` [Variable]

When this variable is true (the default), named instances are saved in binary files as forward references (and `make-load-form` will not be called). If it is false, the normal CLOS instance saving scheme is used. Please note that the default behavior assumes that named objects are properly created (and thus initialized) even when they are loaded from binary files. It is up to the programmer to make sure this happens.

`forget-named-objects` [Function]

This function empties the current namespace of named objects. After this function has been called all named object lookups will fail.

`check-unresolved-forward-references` [Function]
`&optional` *silentp*

This function scans through the current namespace, and collects all forward references. If any are found, a warning is issued (if *silentp* is false, which is the default). This function can be called to find out if there are any forward references left unresolved (in a debugging context, typically). The function returns the forward references found, as a list.

`find-current-object` [Function]
object

If *object* is a named object which has already been purged from the current namespace, this function will return the object that currently resides in the current namespace and has the same name as *object*.

`reinitialize-objects` [Generic function]
thing

Methods of this function replace old (i.e., non-current) named objects "owned" by *thing* with their current counterparts (this mapping can be found out using `find-current-object`). Programmers can define methods for this generic function to "refresh" aggregate data structures (typically after a namespace has been purged and objects have been reloaded). The default method does nothing.

`reinitialize-objects` [Method]
    (*self* `t`)

This method does nothing.

`*named-object-readtable*` [Variable]

This variable holds a readtable which has the added reader macro for reading named objects. After the system has been installed, the value of the variable `*readtable*` is a copy of this readtable.

3.2 Class Definition

If you wish to use the special features described in this manual, you cannot use the standard CLOS class definition form `defclass`; instead, you have to use the new class definition form `defframe`.

`defframe` [Macro]
    name
    supers
    slots
    `&rest` *options*

This definition form is used to define new frame classes. It is similar in form to `defclass`, but does the extra processing needed for frame classes (i.e. ensuring proper metaclass, defining adder, remover and cleaner methods etc.).

The format for slot options is `(`*name key value key value ...*`)` where *name* is the name of the slot, *key*s are slot option keywords, and *value*s are their associated values. Valid slot options are listed below. In addition to these, any CLOS slot options are valid.

`:adder` [Slot option]

This option specifies the name of the value adder function for a slot.

`:remover` [Slot option]

This option specifies the name of the value remover function for a slot.

`:cleaner` [Slot option]

This option specifies the name of the cleaner function for a slot.

`:inverse` [Slot option]

This option specifies the names of the inverse update functions of a slot. The format of the slot option is as follows: (*key name key name* ...) where *keys* are either `:adder`, `:remover` or `:accessor`, and *names* are the names of the corresponding inverse adder, remover and accessor functions. Please note that if you specify an adder and a remover for a slot, you can only specify an inverse adder and remover, but not an inverse accessor; conversely, if you specify an accessor for a slot, you can only specify an inverse accessor, but not an adder nor a remover.

`:allocation` [Slot option]

This standard CLOS slot option has been extended to accept the value `:fragile` in conjunction with many-valued slots. Normally, *copying* list operations are used to maintain the list of values of a many-valued slot. Specifying `:fragile` allocation causes *destructive* list operations to be used in the slot implementation. When `:fragile` is specified one is not allowed to use destructive operations on the value returned by a many-valued slot reader.

Relation slots may not use `:class` allocation.

`:save-values` [Slot option]

This boolean slot option allows one to either save or not save the values of a slot when `make-load-form` generates a load form for binary file instance storage. The default value for this option is `t`. Specifying `nil` for a slot that has an inverse slot causes the values of this slot not to be saved but rather reconstructed by the automatic inverse slot mechanism.

Note that if `*save-named-objects-as-forward-references*` is true, this option has no effect on instances that actually have a name, since `make-load-form` will not be called.

3.3 Instance Creation

`definstance` [Macro]

name
classes
`&rest` *options*

Instances of frame classes can be created just as easily as any other CLOS classes, i.e. using the function `make-instance`. However, to allow one to write declarative instance definitions, the definition macro `definstance` has been created. This definition form creates an instance of a class, and assigns it a name. The *options* are any initialization arguments allowed for *classes*, and are passed directly to `make-instance`.

Currently *classes* may only contain the name of one class. The case of multiple classes is reserved for a possible future extension.

Chapter 4

Metaprogramming Interface

Metaobject protocols are an attempt to solve the perpetual issue of "elegance vs. efficiency" in programming language design by providing a solution which achieves both. The term "metaobject protocol" is understood as a language being implemented as an object-oriented program using itself as the implementation language. Historically the implied introspective and "metacircular" definitions are not uncommon especially when dealing with Lisp. In case of PORK, it being a CLOS program as well as an extension of CLOS, the CLOS metaobject protocol is used as a basis of the corresponding PORK protocol.

PORK has a metaobject protocol (MOP) primarily for creating extensions. If you wish to extend the object system it may be necessary to use this protocol. Ordinary ("non-MOP") users are encouraged to skip this chapter.

4.1 Named Objects

`named` [Class]
`:name` [Initarg]

This class is a base class for named objects, and establishes the named object protocol. It is an abstract class, i.e. not meant to be instantiated. It merely acts as the common superclass for several named object classes uses in the implementation. Programmers are allowed to create subclasses of `named` for their own purposes.

The initialization parameter `:name` can be used to initialize the name of the instance.

`object-name` [Method]
    (*self* `named`)

This method accesses the slot `name`. This slot holds the *name* of an object. In general, it is not guaranteed that the name of an object is stored in a slot in the instance itself. Some other storage method might be used (e.g., a weak hash table). the current implementation uses a slot, however. Internally `object-name` also has a `setf` method, but its purpose is not to change the name of an object (changing object name is not possible at all).

`object-uses-namespace-p` [Generic function]
    thing

This predicate function is used to determine whether a named object should be placed in the current namespace upon creation.

`object-uses-namespace-p` [Method]
    (*self* `named`)

This method returns `nil`. Some subclasses of `named` override this method.

`object-namespace` [Generic function]
    object

This function returns the namespace into which *object* is currently stored, if any.

`object-namespace` [Method]
    (*self* `t`)

This method returns `nil`.

`object-namespace` [Method]
    (*self* `named`)

This method implements the specified functionality of its generic function.

`print-object` [Method]
    (*self* `named`)
    stream

This standard method is called by the Common Lisp system to produce a printed representation of the object *self* into a character stream *stream*. This specific method prints objects using the exclamation-point syntax only if the variable `*print-readably*` is true, otherwise it uses `print-unreadable-object` to print the object; the body of that macro has a call to `print-unreadable-named-object`.

`print-unreadable-named-object` [Generic function]
    thing
    stream
    name

This function is called to print the "body" of an unreadable object printed representation. Programmers are free to specialize this method to provide class-specific information about their objects.

`print-unreadable-named-object` [Method]
    (*self* `named`)
    stream
    name

This method prints *name* (the object's name) into *stream*.

4.2 Namespaces

This section describes the symbol table mechanism used with named objects. Note that none of these functions is normally called by client programs (unless they need to do something special). The previous chapter documents the regular interface to the namespace mechanism.

`namespace` [Class]

This class inherits directly from `named`. This is the class of all namespaces. To create a new namespace, you can call `make-instance` on this class. Namespaces are also named objects, so the name of the namespace is given using the initialization argument `:name`.

`namespace-objects` [Method]
    (*self* `namespace`)

This method accesses the slot `objects`. This slot holds an `eq` -hash table which maps from names to objects.

`namespace-find-named-object` [Generic function]
namespace
name

This generic function performs the name lookup in a namespace and returns the object found, or `nil` if the lookup failed. It is called by `find-named-object`.

`namespace-find-named-object` [Method]
(*self* `namespace`)
name

This method implements the specified functionality of its generic function.

`(setf namespace-find-named-object)` [Generic function]
object
namespace
name

This generic function performs the namespace insertion.

`(setf namespace-find-object)` [Method]
object
(*self* `namespace`)
name

This method implements the specified functionality of its generic function.

`namespace-find-named-objects` [Generic function]
namespace
`&key` *name*
type
predicate

This function can be used for finding objects from namespaces. It returns a list of objects matching the search criteria.

The keyword parameter *name* is used in a call to `namespace-find-named-object`. If *name* is specified, at most one object is returned. The parameter *type* can be any valid Common Lisp type specifier (it defaults to `t`). The parameter *predicate*, if passed, should be a single-parameter predicate function which is used for picking the objects to return.

If multiple criteria are passed (e.g., both a type and a predicate) they all have to match for an object to be returned.

`namespace-find-named-objects` [Method]
  (*self* `namespace`)
  `&key` *name*
    type
    predicate

This method implements the specified functionality of its generic function.

`namespace-forget-named-objects` [Generic function]
  namespace

This method clears the symbol table of a namespace. All lookups fail after this call. This function is called by `forget-named-objects`.

`namespace-forget-named-objects` [Method]
  (*self* `namespace`)

This method implements the specified functionality of its generic function.

`namespace-forward-reference-class` [Generic function]
  namespace

This function returns a class from which the forward references for this namespace are instantiated. The default return value is `forward-reference`.

`namespace-forward-reference-class` [Method]
  (*self* `namespace`)

This method implements the specified functionality of its generic function.

`namespace-unresolved-forward-references` [Generic function]
  namespace

This generic function scans the namespace and returns a list of all forward references found. This function is called by `check-unresolved-forward-references`.

`namespace-unresolved-forward-references` [Method]

(*self* `namespace`)

This method implements the specified functionality of its generic function.

4.3 Forward References

Forward references are unlikely to be specialized in client applications. In some sense, forward references are a mechanism the existence of which one should not have to know about.

`forward-reference` [Class]

This class inherits directly from `named`. This is the base class for all forward references.

`forward-reference-initializations` [Method]
(*self* `forward-reference`)
`(setf forward-reference-initializations)` [Method]
value
(*self* `forward-reference`)

These methods access the slot `initializations`. This slot holds the deferred initializations stored by `add-deferred-initialization`.

`object-uses-namespace-p` [Method]
(*self* `forward-reference`)

This method returns `t`.

`add-deferred-initialization` [Generic function]
object
function
args

This generic function is associated with certain functions (the ones that implement the adder/remover/accessor mechanism) that are called on an object before it gets created (i.e. called on a forward reference). Actual calls to this function are made by an appropriate method of `no-applicable-method` (see the section on relations).

The purpose of this function is to store a function call so that it can be executed later. Typically these are object initializations. The parameter *function* is the generic function the call of which is to be deferred, *args* are the call arguments. If the attempted call is a valid (i.e. deferrable) call, `t` is returned, `nil` otherwise.

`add-deferred-initialization` [Method]
    (*self* `forward-reference`)
    function
    args

This method implements the specified functionality of its generic function.

`add-deferred-initialization` [Method]
    (*self* `t`)
    function
    args

Deferred calls to objects other than forward references are not valid (`nil` is returned). However, calls to null objects are considered valid but are not stored, for those this method just returns `t`.

`update-instance-for-different-class` [Method]
    (*old* `forward-reference`)
    (*new* `frame-object`)
    `&rest` *initargs*

This method calls `initialize-instance` on the *new* object, passing the deferred initializations to the object using keyword parameter `:inits`.

4.4 Frame Objects

`frame-object` [Class]

This class inherits directly from `named`. This is the base class for all frame classes. Unlike the base class `standard-object` which you don't have to specify when using `defclass`, you have to specify this class for `defframe` unless one of the superclasses inherits from it.

`object-uses-namespace-p` [Method]

(*self* `frame-object`)

This method returns `t`.

`shared-initialize` [Method]
(*self* `frame-object`)
slot-names
`&key` *inits*

This method initializes a frame instance by first calling the standard CLOS initialization for ordinary slots, and then calling `relation-shared-initialize` for each of the instance's relation slots. Finally it executes all deferred calls to the instance (passed as the parameter *inits* by the `update-instance-for-different-class` method of `forward-reference`).

For `frame-object` to work properly, this method must *never* be overridden.

`update-instance-for-different-class` [Method]
(*old* `forward-reference`)
(*new* `frame-object`)
`&rest` *initargs*

This method exists because `change-class` does not take any initialization parameters. This method just calls the next method with initargs that were stored in a special variable by `make-instance` (see the section on implementation notes).

4.5 Frame Classes

`frame-class` [Class]

This class inherits directly from `standard-class`. This is the metaclass of all frame classes. The definition macro `defframe` automatically places `frame-class` in the CLOS class definition as a metaclass. Please note that just specifying `frame-class` as your metaclass is not enough, you have to use `defframe`.

`class-direct-relations` [Method]
(*self* `frame-class`)
`(setf class-direct-relations)` [Method]
value

(*self* `frame-class`)

These methods access the slot `direct-relations`. This slot holds the *direct* (i.e. local) relations of a class, as a list of relation definitions.

`class-relations` [Method]
    (*self* `frame-class`)
`(setf class-relations)` [Method]
    value
    (*self* `frame-class`)

These methods access the slot `relations`. Class finalizations collects all relations definitions of a class and its superclasses into this slot.

`class-saved-relations` [Method]
    (*self* `frame-class`)
`(setf class-saved-relations)` [Method]
    value
    (*self* `frame-class`)

These methods access the slot `saved-relations`. This slot holds a list of relation names to be saved by `make-load-form`.

`class-saved-slots` [Method]
    (*self* `frame-class`)
`(setf class-saved-slots)` [Method]
    value
    (*self* `frame-class`)

These methods access the slot `saved-slots`. This slot holds a list of slot names to be saved by `make-load-form`.

`make-instance` [around method]
    (*class* `frame-class`)
    `&key` *name*

This method handles name lookup and forward reference coercion of frame objects. Four different cases of object creation or reinitialization can be identified, based on the result of namespace lookup on *name*:

Nothing - the inherited `make-instance` is called.

Forward reference - it is coerced to the desired type (by calling `change-class` which handles initialization).

Old instance, same class - initialization by `reinitialize-instance`.

Old instance, different class - a continuable error is issued, after which the object is coerced to the desired type (by calling `change-class`).

`compute-saved-relations` [Generic function]
    class

This generic function returns two values: a list of names of relations which are to be saved (by `make-load-form`) and a list of named of slots which are to be saved. This function is called during class finalization.

`compute-saved-relations` [Method]
    (*class* `frame-class`)

This method implements the specified functionality of its generic function.

`initialize-frame-class` [Generic function]
    class
    relations

This generic function is called to initialize a frame class. This happens between the definition of the class and the creation of the first instance of the class. The parameter *relations* is a list of relation definition instances.

`initialize-frame-class` [Method]
    (*class* `frame-class`)
    relations

This method implements the specified functionality of its generic function.

`initialize-frame-class` [Method]
    (*class* `symbol`)
    relations

This method calls `initialize-frame-class` on the result of class lookup on the symbol *class*.

4.6 Relations

For reasons of portability we do not assume full CLOS MOP. Instead, metaobjects for relation slots have been kept separate from any possible standard slot definition metaobjects.

`relation-definition` [Class]
`:initargs` [Initarg]
`:initfunction` [Initarg]
`:reader` [Initarg]
`:save-values-p` [Initarg]

This class inherits directly from `named`. This is the base class for relation slot metaobjects, i.e. the objects describing the relation slots of a class.

`relation-definition-initargs` [Method]
    (*self* `relation-definition`)

This method accesses the slot `initargs`. This slot holds the names of the initialization keyword parameters of the slot/relation.

`relation-definition-initfunction` [Method]
    (*self* `relation-definition`)

This method accesses the slot `initfunction`. This slot stores the *initialization function* of the slot/relation. The function is a lambda of zero arguments, created in the correct lexical environment of the definition form. Calling the function produces the initial value of the slot.

`relation-definition-reader` [Method]
    (*self* `relation-definition`)
`(setf relation-definition-reader)` [Method]
    value
    (*self* `relation-definition`)

These methods access the slot `reader`. This slot holds the name of the reader function of the slot.

`relation-definition-save-values-p` [Method]
    (*self* `relation-definition`)

This method accesses the slot `save-values-p`. This slot has a boolean value indicating whether the values of this relation should be saved by the code generated by `make-load-form`.

`relation-shared-initialize` [Generic function]
object
relation
`&optional` *value*

This generic function is called by the object initialization code (the generic function `shared-initialize` for `frame-object`). The purpose of this function is to initialize the slot/relation correctly by calling the appropriate writer or adder function.

`sv-relation-definition` [Class]
`:writer` [Initarg]

This class inherits directly from `relation-definition`. This is the class for single-valued relations.

`relation-definition-writer` [Method]
(*self* `sv-relation-definition`)
`(setf relation-definition-writer)` [Method]
value
(*self* `sv-relation-definition`)

These methods access the slot `writer`. This slot holds the name of the writer function of the relation.

`relation-shared-initialize` [Method]
object
(*relation* `sv-relation-definition`)
`&optional` *value*

This method implements the specified functionality of its generic function.

`mv-relation-definition` [Class]
`:adder` [Initarg]
`:remover` [Initarg]
`:cleaner` [Initarg]

This class inherits directly from `relation-definition`. This is the class for many-valued relations.

`relation-definition-adder` [Method]
  (*self* `mv-relation-definition`)
`(setf relation-definition-adder)` [Method]
  value
  (*self* `mv-relation-definition`)

These methods access the slot `adder`. This slot holds the name of the adder function of the relation.

`relation-definition-remover` [Method]
  (*self* `mv-relation-definition`)
`(setf relation-definition-remover)` [Method]
  value
  (*self* `mv-relation-definition`)

These methods access the slot `remover`. This slot holds the name of the remover function of the relation.

`relation-definition-cleaner` [Method]
  (*self* `mv-relation-definition`)
`(setf relation-definition-cleaner)` [Method]
  value
  (*self* `mv-relation-definition`)

These methods access the slot `cleaner`. This slot holds the name of the cleaner function of the relation.

`relation-shared-initialize` [Method]
  object
  (*relation* `mv-relation-definition`)
  `&optional` *values*

This method implements the specified functionality of its generic function.

`relation-generic-function` [Class]

This class inherits directly from `standard-generic-function`. This is the meta-class of all accessor, adder, remover and cleaner functions of frame objects.

Note: the accessor generic functions of slot relations differ from ordinary CLOS accessor functions by having an optional parameter *updatep* (writer, adder and remover functions). This parameter is only used internally. It is an error to specify a value for it when calling the accessor methods. It is included in the documentation because you need to specify it when defining additional (e.g. `:before`, `:after` etc.) methods for the accessor functions.

`no-applicable-method` [Method]
  (*function* `relation-generic-function`)
  `&rest` *args*

This method calls `add-deferred-initialization` to store the (apparently) "premature" call to *function*. If that function returns `nil` (no deferred calls possible), this method calls the next method, effectively signalling an error.

Chapter 5

Design and Implementation Notes

This chapter will provide more details about the current implementation of PORK, especially about features the implementation of which is important to understand in order to use these features correctly.

5.1 Object Naming Mechanism

One of the design issues with named objects was whether instance names should be stored with the instances themselves or in a centralized repository. Typical situation when using named instances is that some of the instances created by one's program are named and some are not. If the name is always stored with the instance itself, a slot has to be allocated for this purpose, even in those instances which are anonymous.

The storage of instance names in a centralized repository (such as a hash table) leads to another problem. If there is mapping involving the instances, there are also references to these instances. This hinders the operation of automatic storage reclamation. Same problem occurs of course with the name-to-object mapping, but these references to instances are destroyed when a namespace is "emptied" (e.g., when a model is reloaded); in the case of the other mapping this is not the case, because we may want to know the name of an object even though it has been removed from the namespace (especially in the context of the `reinitialize-objects` protocol). In those Common Lisp systems where *weak pointers* are available for use with hash tables (e.g., the Macintosh Common Lisp [20]) these can be used to allow the garbage collector reclaim instances to which the only references are through the symbol table. This feature is not available as part of the Common Lisp standard, making its use questionable.

The centralized repository with weak pointer was experimented with, but the problem with the "unremovable" object references is actually a lot bigger problem than an additional (unused) slot in every instance: the variations in performance and memory consumption in the DITOPS scheduling system were almost negligible (in tests where the system created very large numbers of anonymous objects).

5.2 Forward References and Deferred Access

The `!` character syntax expands to a call to `ensure-named-object` which, after failing to find an object with the given name in the current namespace, will create a forward reference with a call to `make-instance`, passing `forward-reference` as the class and the name using the keyword parameter `:name`.

If a forward reference is added to a slot which has an inverse slot the call to update the inverse slot has to be deferred. The `make-instance` call in section 2.3 creating a time line for a resource can serve as an illustrative example. We will assume that the resource `!drill-1` does not exist when the reader sees the call, thus a forward reference `!drill-1` is created. The definition for the *writer* portion of the accessor generated looks (approximately) like this:

```
(defmethod (setf resource) (resource (self time-line)
                             &optional updatep)
  (let ((r (resource self)))
    (when r
      (setf (time-line r) nil))
    (unless updatep
      (setf (time-line resource t) self))
    (setf (slot-value self 'resource) resource)))
```

During initialization the writer method is called, and the forward reference named `drill-1` gets passed as the parameter *resource*. Thus a call to `time-line` is made with this forward reference as the first parameter. Since forward references do not have a method for this function, CLOS calls the generic function `no-applicable-method`. All relation accessor generic functions (in this case the function `time-line`) are instances of `relation-generic-function` which has a method for `no-applicable-method`. This method stores all relevant information about the original call to `time-line` (by calling `add-deferred-initialization`). This way the call gets recorded even if the actual resource does not exist.

When a `make-instance` call is made to create the resource named `drill-1`, the method (of metaclass `frame-class`) finds the forward reference and subsequently calls the function `change-class` to coerce it to the appropriate class, after which `change-class` calls `update-instance-for-different-class`. There is a method for this function to change forward references to frame objects. The default method (of `standardobject`) would call `shared-initialize`. With PORK it was decided that the new method should call `initialize-instance` because this would be intuitive to the client programmer.

The real problem arises from the fact that there are no parameters that would allow the original *initargs* to be communicated this far down in the protocol. Therefore the `make-instance` method of `frame-class` binds the *initargs* to a special variable, and the `update-instance-for-different-class` method of `forward-reference` uses this binding when calling `initialize-instance`. It should be noted that the Common Lisp standardization committee has added the provision of passing *initargs* to change class to the proposed ANSI Common Lisp standard [22, chap.7 p.38].

5.2.1 Named Instances and Binary Files

In standard CLOS, instances saved to binary files are reconstructed by a "two-step" process: first, the instance is created; second, any slot values the instance had when it was saved are restored. In PORK a slightly different scheme has been adopted (mainly because of forward references): all named instances are saved as forward references. When they get loaded (that is, when references to them get loaded) the normal forward reference resolution process is invoked. This assures that references will always be correct as far as names are concerned (and names are important, why else would one be using them). The "flipside" of this scheme is that one has to make sure that any named instance -creating forms in one's program are executed when the program is loaded from binary files.

5.3 Class Definition

The metaclass `frame-class` does not have a major role in the definition of new classes. The expansion function for the definition macro `defframe` calls `make-slot-spec` for each slot specification to produce three things:

- A CLOS-conformant slot specification. This slot implements the PORK relation slot in question.
- Method definitions for the special PORK accessor functions (adders, removers and cleaners, accessors capable of handling inverse slots).
- A form which, when evaluated, yields the relation metaobject for the slot.

The definition macro expands into a `progn` containing a check for a proper metaclass, the `defclass` form, the access method definitions, and a call to `initialize-frame-class` with the relation metaobject forms as parameters. The method `initialize-frame-class` for `frame-class` stores the relation metaobjects into the class metaobject. These are only the direct relation metaobjects, the full list of relations is computed during class finalization by an `:after` method of `finalize-inheritance`. This method also caches a list of the slot and relations to be saved by `make-load-form`.

The access method definitions for relation slots are constructed by `make-slot-spec`. For many-valued slot value deletion, either the function `remove` or the function `delete` is used, depending on the value of the `:allocation` slot option. The definition of accessor methods and their optional parameter have been discussed in section 2.4.1.

5.3.1 Class Precedence List

Analogously to CLOS classes inheriting from `standard-object`, objects defined using `defframe` have to inherit from `frame-object`. It would be nice to automate this (that is, to automate the insertion of the base class into the class precedence list), but without the full MOP this proves to be difficult. It is not enough to just define a new method for `compute-class-precedence-list`, because `frame-object` would also possibly have to made a direct superclass of a class being defined. The definition macro could do this, but since at definition time it is not known whether this has to be done, the only possible solution is to insert it at the end of the direct superclass list of *all* classes defined using `defframe`. This was experimented with at first, and it works fine *per se*, but produces really bizarre results with class browsers (it tends to flatten the class tree since every frame class is now a direct subclass of `frame-object`).

5.4 Instance Creation and Initialization

The interesting detail in instance creation is how to make CLOS instance initialization and PORK relation initialization coexist. In CLOS, the slot initialization is handled by the `shared-initialize` method of `standard-object`. In PORK, relation initialization is handled by a method of the same generic function, specialized for `frame-object`. These are the steps taken to initialize a frame instance:

1. Ordinary slots and relation slots are separated. The list of slot names passed to `shared-initialize` is left untouched, but the relation *initargs* are removed from the list of initargs.
2. The default initialization is called. Relation slots that have not been initialized are left unbound.
3. The function `relation-shared-initialize` is called on each relation to be initialized. Methods for this function exist for both single-valued and many-valued relations.
4. Any deferred calls are executed (this is discussed in section 5.2).

Single-valued relation slots (that is, those that have an inverse slot; if there is no inverse slot, ordinary CLOS initialization is used) are initialized by calling the writer method of the slot. This way the inverse slot update mechanism is given a chance to handle the inverse slot, and also any daemons defined for the slot will run.

Many-valued relation slots are first "cleaned" (either by setting them directly to `nil`, running their cleaner method, or calling their remover repeatedly). The initial values are then added to the slot, one by one, by calling the relation adder method repeatedly.

5.5 Common Lisp System Compatibility

PORK needs to address two specific categories of compatibility with Common Lisp systems:

- The implementation needs to be compatible with standard "CLtL2" -type Common Lisp [17], possibly taking care of minor differences between implementations. Support for "CLtL1" Common Lisps was provided with early versions of PORK but has since been dropped. Some changes introduced by the ANSI Common Lisp have been anticipated and the source code "read-conditioned" using `#+:x3j13`.
- The implementation needs to be compatible with a subset of the proposed Common Lisp Object System Metaobject Protocol (CLOS MOP) [6]. Reliance on concepts defined in this proposal is described in section 5.6.

Since PORK is an extension of CLOS it really makes little sense to provide full "CLtL1" support. Because of the scarcity of fully conformant "CLtL2" Common Lisps at the time when PORK was initially designed we have adopted a "relaxed" definition of "CLtL2" (particularly since we wanted to support Lucid Common Lisp). The specific "CLtL2" functions and features we require are:

- "ANSI-conformant" packages, primarily the package `COMMON-LISP`,
- logical pathnames (preferred but not absolutely necessary),
- macros `destructuring-bind` and `print-unreadable-object`,
- definition macro `defpackage`, and
- the function `make-load-form-saving-slots`.

PORK is intended to be constructed using the make-system tool written by Mark Kantrowitz at CMU [4], but since PORK only consists of very few source files even manual system construction is very easy.

5.6 Reliance on Common Lisp MOP

The reliance on the CLOS MOP [6] has been minimized in the current version of PORK. The CLOS MOP is not an official part of the Common Lisp standard proposal [17], thus variations in details and completeness of implementations exist between different Common Lisp systems. Mainly PORK relies on what could be called the "introspective" subset of CLOS MOP, using functions which allow existing metaobject structures to be inspected. PORK also uses the *Class Finalization Protocol* for constructing the relation structures needed by frame classes. In an early design phase the use of the *Instance Structure Protocol* was considered but later abandoned because of two reasons:

1. The Macintosh Common Lisp (MCL) does not support the Instance Structure Protocol and extending it to do so may result in "dangerous" and/or inefficient implementations.
2. Because of total reliance on slot accessor functions, all slot accesses can be implemented on top of the standard `slot-value` function. It should be noted, however, that a CRL/BEEF -type context mechanism might be easiest to implement using the Instance Structure Protocol. This mechanism may be added in the future.

PORK's main use of the CLOS MOP is to provide introspective access to the implementation. The use of the so-called "introspective MOP" is minimal, though. Here are the functions employed by PORK:

- `class-direct-superclasses`,
- `class-name`,
- `class-slots`,
- `class-precedence-list` and
- `slot-definition-name`.

PORK also employs the Class Finalization Protocol by calling the following generic functions or defining methods for them:

- `validate-superclass`,
- `finalize-inheritance`,
- `class-finalized-p` and
- `compute-class-precedence-list`

5.6.1 MOP and Macintosh Common Lisp

Macintosh Common Lisp (version 2.0) [20] does not support the full proposed CLOS MOP. Fortunately it supports an "introspective" subset of MOP [21]. To port PORK to MCL the required remaining parts of the MOP had to be added.

The technique used in adding MOP functions works as follows: let's say a functionality that should be provided by the MOP function A is provided in MCL by function B. We redefine B to call A the definition of which calls the *original* definition of B. By "sandwiching" functions in this manner we can have the appropriate MOP functions be called at appropriate times. Implementations which then override or specialize these functions will work as expected.

The missing functions of the Class Finalization Protocol were defined as follows:

- The function `finalize-inheritance` was implemented by "sandwiching" the internal function `ccl::initialize-class-and-wrapper`. This function calls a local recursive function so a series of recursive calls to superclasses has to be made separately.
- The function `compute-class-precedence-list` was implemented by "sandwiching" the internal function `ccl::compute-cpl`.
- The function `class-finalized-p` was implemented as follows:

  ```
  (defmethod class-finalized-p ((class class))
    (not (null (slot-value class 'ccl::own-wrapper))))
  ```

- The function `validate-superclass` was written according to the specification in the CLOS MOP [6]. MCL has no counterpart.

A more detailed account on the implementation of the MOP extension of MCL is given in [11].

5.6.2 MOP and Other Common Lisps

To make PORK run on TI Explorer Common Lisp (TICL), certain additional definitions were required. TICL treats generic function objects differently, thus the method of `no-applicable-method` specialized for `relation-generic-function` would never get called. The following definition was required for the class `symbol` (and a similar one for `cons`):

```
(defmethod no-applicable-method :around ((function symbol)
                                          &rest args)
  (apply #'no-applicable-method
         (ticlos::get-generic-function-object function)
         args))
```

For both TICL and Lucid Common Lisp (version 4.1) the class finalization code (generic function `finalize-inheritance`) had to be called explicitly (the method `make-instance` of class `frame-class`). For Lucid CL the real class finalization happens "too soon", thus a substitute finalization protocol was defined to take care of relation metaobjects, and called explicitly from `make-instance`.

5.7 Porting to Other Platforms

PORK has been designed to be portable between Common Lisp systems. Section 5.5 addresses the compatibility issues of Common Lisp systems from PORK's standpoint, and section 5.6 addresses issues of MOP reliance (MOP being the only potentially problematic source of incompatibility).

One of the differences between Common Lisp systems is their package structure. Different implementations have their implementation-dependent code in different packages; MOP functions are often purely "internal" to an implementation. The package definition of PORK shields the rest of the PORK implementation from these differences (read-time conditionals have been used to differentiate between different implementations). When porting PORK to a new platform, one invariably has to change the package definition.

Bibliography

[1] Richard Fikes and Tom Kehler, 1985. "The Role of Frame-Based Representation in Reasoning", *CACM*, 28(9) 904-920.

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

[3] Juha Hynynen and Ora Lassila, 1989. "On the Use of Object-Oriented Paradigm in a Distributed Problem Solver", *AI Communications*, 2(3) 142-151.

[4] Mark Kantrowitz, 1991. *Portable Utilities for Common Lisp, User Guide and Implementation Notes*, Report CMU-CS-91-143, Pittsburgh (PA), School of Computer Science, Carnegie Mellon University.

[5] Sonya E. Keene, 1989. *Object Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Reading (MA), Addison-Wesley.

[6] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow, 1991. *The Art of the Metaobject Protocol*, Cambridge (MA), MIT Press.

[7] Gregor Kiczales and John Lamping, 1992. "Issues in the Design and Specification of Class Libraries", in *OOPSLA'92 Conference Proceedings*, ACM Sigplan Notices 27(10) 435-451.

[8] Ora Lassila, 1990. "Frames or Objects, or Both?", in *Workshop Notes from the 8th National Conference on Artificial Intelligence (AAAI-90): Object-Oriented Programming in AI*, Boston (MA), AAAI. [Report HTKK-TKO-B67, Otaniemi (Finland), Department of Computer Science, Helsinki University of Technology].

[9] Ora Lassila, 1991. *BEEF Reference Manual – A Programmer's Guide to the BEEF Frame System* (second version), Report HTKK-TKO-C46, Otaniemi

(Finland), Department of Computer Science, Helsinki University of Technology.

[10] Ora Lassila, 1992. *The Design and Implementation of a Frame System*, Otaniemi (Finland), Faculty of Technical Physics, Helsinki University of Technology, Master's Thesis.

[11] Ora Lassila, 1992. "Oliojärjestelmän laajentaminen metaobjektiprotokollan avulla" ("Extending an Object System using a Metaobject Protocol", in Finnish), unpublished report, Otaniemi (Finland), Department of Computer Science, Helsinki University of Technology.

[12] Marvin Minsky, 1975. "A Framework for Representing Knowledge", in *The Psychology of Computer Vision*, Patrick Henry Winston (ed.), New York (NY), McGraw-Hill.

[13] Smith, S.F., 1993. "OPIS: A Methodology and Architecture for Reactive Scheduling", in *Intelligent Scheduling*, (eds. M. Fox and M. Zweben), Morgan Kaufmann Publishers.

[14] Stephen F. Smith and Katia P. Sycara, 1993. "A Constraint-Based Framework for Multi-Level Management of Transportation Schedules", San Antonio (TX), DARPA Planning Workshop.

[15] Stephen F. Smith and Ora Lassila, 1994. "Configurable Systems for Reactive Production Management", in *Knowledge-Based Reactive Scheduling*, IFIP Transactions B-15, Amsterdam (The Netherlands), Elsevier Science Publishers.

[16] Stephen F. Smith and Ora Lassila, 1994. "Toward the Development of Flexible Mixed-Initiative Scheduling Tools", in *ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative Workshop Proceedings*, Tucson (AZ), Morgan Kaufmann, pp.145-154.

[17] Guy L. Steele, Jr., 1990. *Common Lisp – the Language* (second edition), Bedford (MA), Digital Press.

[18] –, 1985. *KEE Software Development System User's Manual*, Mountain View (CA), Intellicorp, Inc.

[19] –, 1986. *Knowledge Craft User's Manual*, Pittsburgh (PA), Carnegie Group, Inc.

[20] –, 1991. *Macintosh Common Lisp 2.0 Reference*, Draft 030-5008-A, Cupertino (CA), Apple Computer, Inc.

[21] –, 1992. *MCL's Metaobject Protocol*, internal document, Apple Computer, Inc.
(ftp://cambridge.apple.com/pub/MCL2/docs/introspective-mop.txt)

[22] –, 1994. *X3.226-199x, Programming Language Common Lisp* (Second Public Review and Comment Period document), Washington (DC), X3 Secretariat, American National Standards Institute.

Index

(setf class-direct-relations), 38
(setf class-relations), 39
(setf class-saved-relations), 39
(setf class-saved-slots), 39
(setf forward-reference-initializations), 36
(setf namespace-find-named-object), 34
(setf namespace-find-object), 34
(setf relation-definition-adder), 43
(setf relation-definition-cleaner), 43
(setf relation-definition-reader), 41
(setf relation-definition-remover), 43
(setf relation-definition-writer), 42
named-object-readtable, 28
namespace, 26
save-named-objects-as-forward-references, 27
:adder, 28, 42
:allocation, 19, 29
:cleaner, 29, 42
:initargs, 41
:initfunction, 41
:inverse, 29
:name, 31
:reader, 41
:remover, 28, 42
:save-values, 18, 29
:save-values-p, 41
:writer, 42

add-deferred-initialization, 36, 37
adder, 8, 11
allocation of slots, 19

BEEF, well done, 1

check-unresolved-forward-references, 27
class precedence list, 48
class, defining a, 28, 48
class, specified, 20
class-direct-relations, 38
class-relations, 39
class-saved-relations, 39
class-saved-slots, 39
CLOS, 7, 50, 51
Common Lisp, 50
compute-saved-relations, 40

daemon, 15
deferred access, 46
'defframe, 28
definstance, 29

ensure-named-object, 26

find-current-object, 27
find-named-object, 26
find-namespace, 26
forget-named-objects, 27
forward reference, 5, 36, 46
forward-reference, 36
forward-reference-initializations, 36
frame-class, 38
frame-object, 37

initialization, 20, 29, 49
initialize-frame-class, 40
inverse relation, 9

make-instance, 39
many-valued slot, 8
metaobject protocol, 20, 31, 51
mv-relation-definition, 42

named, 31
named object, 4, 25, 31, 45
namespace, 21, 25, 33, 45
namespace-find-named-object, 34
namespace-find-named-objects, 34, 35
namespace-forget-named-objects, 35
namespace-forward-reference-class, 35
namespace-objects, 33
namespace-unresolved-forward-references, 35
no-applicable-method, 44

object-name, 26, 32
object-namespace, 32
object-uses-namespace-p, 32, 36, 37

PORK, 1
print-object, 32
print-unreadable-named-object, 33

reinitialize-objects, 27, 28
relation, 6, 8, 9, 41
relation-definition, 41
relation-definition-adder, 43
relation-definition-cleaner, 43
relation-definition-initargs, 41
relation-definition-initfunction, 41
relation-definition-reader, 41
relation-definition-remover, 43
relation-definition-save-values-p, 41
relation-definition-writer, 42
relation-generic-function, 43
relation-shared-initialize, 42, 43
remover, 8, 11

shared-initialize, 38
slot options, 18
slot, allocation of a, 19
slot, saving values of a, 18
specified class, 20
sv-relation-definition, 42

update-instance-for-different-class, 37, 38