Dynamic Persistent Metadata: A Metaobject Protocol Based Approach to Increasing Power in Knowledge Representation

Eric L. Peterson*
Artificial Intelligence Technical Center
MITRE Corporation
7525 Colshire Dr.
McLean, VA 22102-3481
e-mail: eric@ai.mitre.org
URL: http://www.cs.umd.edu/users/ericp
phone: (703) 883-6116
FAX: (703) 883-6435

July 27, 1995

Abstract

Object-oriented and relational databases typically provide some sort of metadata, such as the documentation and type information for slots or columns. The tool suite described in this paper goes beyond this typical set of services by allowing application programmers and software tool builders of knowledge representation intensive systems (*i*) to extend this usual set of metadata to allow for other metadata such as unit of measure or relation type, (*ii*) to associate a non-typical variety of metadata with a slot on a particular instance such as a slot state history or a rating of the slot contents' probability of correctness, (*iii*) to allow that these metadata types and their values be dynamically added, deleted and altered, and (*iv*) to allow this metadata to persist and therefore be available for subsequent sessions. The implementation is based on the Metaobject Protocol (MOP) of the Common Lisp Object System (CLOS). The MOP allows for the seamless extension of CLOS's object-oriented functionality without alteration of any CLOS source code.

1 Introduction

Object-oriented and relational databases typically provide some sort of metadata, such as the documentation and type information for slots or columns. The tool suite described in this paper goes beyond this typical set of services by allowing application programmers and software tool builders of knowledge representation intensive systems (*i*) to extend this usual set of metadata to allow for other metadata such as unit of measure or relation type, (*ii*) to associate a non-typical variety of metadata with a slot on a particular instance such as a slot state history or a rating of the slot contents' probability of correctness, (*iii*) to allow that these metadata types and their values be dynamically added, deleted and altered, and (*iv*) to allow this metadata to persist and therefore be available for subsequent sessions. The implementation is based on the Metaobject Protocol (MOP) of the Common Lisp Object System (CLOS). The MOP allows for the seamless extension of CLOS's object-oriented functionality without alteration of any CLOS source code.

[0]Eric Peterson is involved in machine translation at the Artificial Intelligence Technical Center at The MITRE Corporation. Special thanks go to Fernando Mato Mira for his review and suggestions, Dr. Larry Kerschberg for his suggestions and encouragement, to an anonymous reviewer for suggestions.

Shared facets provide the ability to associate and retrieve information having to do with a particular slot of a given class. In database circles this is known as metadata; I will refer to this data as shared facet data or shared metadata (as opposed to instance facet data which can vary from instance to instance). Shared metadata does not vary over the instances of a class.

Instance metadata, on the other hand, provides the ability to associate and retrieve information that belongs to a particular slot of a given instance of a class. I will call this information instance facet data or instance metadata — as opposed to shared slot facet data, which applies to a particular slot of a given class over all its instances. This instance slot facet data, on the other hand, can vary over the instances of the class. By means of such functionality, the applications programmer may create and later request an instance facet such as *slot-state-history* for a particular slot of a given class. Obviously, for a given slot, each instance would need its own history. Therefore this type of facet would be the right choice for a slot state history.

2 Motivation for this Work

In spite of the fact that facets are not a terribly new idea, users of CLOS [Steele] do not have facet functionality available in persistent form. In fact, until very recently there has been no CLOS-based public domain or commercial implementation of either kind of facets with or without persistence. The following subsections detail some of the advantages and uses of persistent facets.

2.1 Shared Metadata

For a large part of the many possible database applications requiring metadata, the metadata values need not ever change. For example, the writers of an application may decide that they want their measurement data to be exclusively in the metric system. By themselves, then, the application writers have no need for a shared facet for every one of their measurement slots to indicate that the unit of measure is metric. If they are to make their data public, however, such unit of measure metadata would be invaluable for informing their clients of the measurement representation conventions. This tool services such unchanging metadata needs with shared and instance metadata capabilities, but the persistent aspect of the shared metadata tool is not necessary in this case because the unchanging metadata values are specified in class definitions. These values would be loaded with the class definitions.

The remaining applications require that their metadata be programatically changed as needed. For example, with a database consistency tool, certain databases update other databases according to appropriate constraints. A weapons officer aboard a ship may wish to change the frequency of updates to weather-related slots if ship-launched weapons are sensitive to weather conditions. Because fresher data is more accurate, the needed frequency of updates would, of course, depend on the ship's likelihood of needing to launch weapons. These varying consistency constraints would need to persist and would very naturally, in the present example, reside in shared facets of the *wind-speed* weather-related slots.

Examples of other uses for shared metadata are as follows:

- *Database consistency management:* To continue with the consistency example, the following constraints were also implemented in a non-persistent version of this metadata tool suite. In a deployed environment, however, these constraints would have to be persistent. These constraints are completely described in [Seligman] .
  - *Selection conditions:* These constraints specify conditions under which a cached instance will be created at a remote database. The constraints that only have to do with the value of a slot are naturally stored as shared metadata.
  - *Retraction conditions:* These constraints specify conditions under which a cached instance will be purged from a client database. The constraints that only have to do with the value of a slot are naturally stored as shared metadata.
  - *Refresh attributes:* This metadata item stores an indication of whether that instance slot will be updated when an instance update occurs.

- *Relation type:* I have found it useful to annotate slots referring to other objects with a specification of the type of relation involved. For example, in an automobile object, a slot containing a list of wheel objects would be labeled as a *:compositional* slot, because, in part, the car is composed of tires. In spreading activation search, this relation type information served in certain cases to prune the search to parts of the object in question and excluded arbitrarily related data.[1]

- *Daemon behavior:* In the implementation of certain types of inference engines, it is useful to associate triggering behavior with slots of a certain class. Because usually it is undesirable to trigger inferencing off of every single slot state change, certain key slots can be chosen as *active slots*. They are active in the sense that changes to their values will trigger such inferencing. A shared facet very easily could note whether a slot is active.

- *Slot display criteria:* Under certain conditions, it is not desirable to display the contents of all slots of a class when displaying an object. A farm equipment application may not want to display inherited utility slots for things such as the unique database id number of a reaper object.

- *Instance facet requests:* In the tool suite described in this paper, shared facets are used to help implement instance facets. To best understand this usage of shared facets requires knowledge of the instance facet implementation explained later in the paper. Suffice it to say that a given slot and its instance facets are implemented by an instance of a custom mixed class. This class's metaobject is stored in a shared facet associated with the slot in question.

2.2 Instance Metadata

Instance facet data, by definition, must be allowed to vary with every instance, so reading in the data from a class definition is not a viable option. Regardless of whether the metadata is to change after the point that it is originally specified, persistence is still required to capture the final state of the metadata.

Examples of uses for instance metadata are as follows:

- *Source documentation:* A facet could contain text annotation regarding where the slot information came from. This information could also be in machine discernible form.

- *Confidence level:* A facet could contain an assessment of the probability that the slot's information is correct.

- *Multiple hypotheses:* When the slot value is not known, a facet could contain multiple hypotheses about the slot's value.

- *Margin for error:* A facet could contain upper and lower bounds for possible error.

- *Slot state history:* The facet could contain a history of all prior state changes of the slot along with the corresponding time-stamps.

- *Truth maintenance dependencies:* The facet might contain a list of values from which the slot's value was derived.

- *Unit of measure:* Although unit of measure information could be implemented globally over the instances of a given class's slot, the user might just as well want the option of having people's heights entered in the unit of measure of her choice.

- *Expected future values:* Computed expected future values of a slot could be cached in an instance facet.

3 Technical Rationale

The chief implementation decisions are explained as follows:

[1] Spreading activation search often involves a naive breadth first search of a semantic network.

```
(defmetaclass unit-of-measure-class
  (shared-slot-facets-class)
  ()
  (:slot-definitions-mixin-slots
   (unit-of-measure
    :accessor unit-of-measure
    :initarg :unit-of-measure
    :initform nil)))
```

```
(defclass person (mammal)
  ((employer
    :documentation
    "The company for which the person works.")
   (height
    :documentation "Height of person."
    :unit-of-measure :centimeters)
   ... ;other slots
   )
  (:metaclass unit-of-measure-class))
```

Figure 1: The *defclass* definition utilizes CLOS-like *:keyword* slot option syntax for shared unit of measure metadata in a definition of *person* class. This figure assumes that mammal inherits from *unit-of-measure-object* just as one assumes that it is not necessary to explicitly inherit *standard-object* into every object of metaclass *standard-class*. The definition of *unit-of-measure-class* precedes its usage for completeness. The *defmetaclass* macro is explained later. The *unit-of-measure-object* is simply a subclass of *standard-object* and has a metaclass of *unit-of-measure-class*.

3.1 Use of the CLOS MOP

3.1.1 Seamless extendibility of CLOS

Since this system is developed in CLOS, its MOP was available to aid in the extension of CLOS to provide facets. The MOP is simply a useful subset of the classes and generic functions that implement CLOS's semantics[2]. These classes can be subclassed to allow the CLOS extension implementor to add slots to the subclasses of these metaclasses. The behavior of the generic functions can be augmented or, in some cases, replaced by the addition of methods that add to or completely mask the methods provided by CLOS. These methods specialize on and are associated with the CLOS-extending subclasses. All CLOS extensions are made using basically just these techniques. Using these methods, there is no need to change or even possess CLOS source code to extend its functionality.

Because of the MOP's ability to lay open the internal implementation of CLOS, it was possible to subclass CLOS's metaclasses and extend the behavior of key MOP methods in a way that is virtually seamless and intuitive to the CLOS user. With the extensions described in this paper, CLOS still retains all its inherent behavior. The user simply utilizes extra keyword parameters in slot definitions and specifies a different metaclass (See Figure 1).

3.1.2 Reflection

CLOS can essentially query itself via the MOP application programmer interface and find out virtually anything about its own runtime state. The program need only perform simple slot access on the *metaobjects* that implement it.[3] A class metaobject can be queried to find out its slots, superclasses, subclasses, etc. A slot can be queried in order to discover, among other things, its type or documentation string. This ability for CLOS programs to reason about or *reflect* on the object-oriented aspects of their own workings is properly referred to as reflection.[4]

This reflective power comprises much of the foundation upon which these facets are built. Without going into a burdensome level of detail, suffice it to say that facet code needs and uses introspective calls such as the following: CLASS-DIRECT-SLOTS, CLASS-DIRECT-SUPERCLASSES, CLASS-OF, CLASS-PROTOTYPE, CLASS-SLOTS, SLOT-DEFINITION-NAME.

These facets, in turn, attempt to *reimburse* CLOS and its MOP for all these introspective helps by

[2] Generic functions have one or more methods associated with them. Each method is usually associated with a particular class, but can be associated with multiple classes. Generic functions *decide* which of their methods to call depending on the class(es) of the generic function's required argument(s).

[3] This, of course, assumes that metaobject attributes are implemented as slots. They need not be impemented as slots

[4] [Maes and Nardi] define reflection as follows: "A computational system is said to be reflective when it is itself part of its own domain."

extending CLOS's reflective power. Instead of being restricted to querying a slot about a couple of semantic aspects such as type and documentation string, and programatic aspects such as the names of its accessors, the user can now query a slot about as much slot related information as there are shared facets associated with the slot. Both semantic information concerning the thing that the slot represents (such as unit of measure information) and programatic/utilitarian information (such as which slots are to be displayed) can be richly extended, queried and altered.

Instance facets provide not an extension of existing slot introspective power, but rather an introduction of a further kind of introspective power - the ability to query individual instances about the nature of the information in their slots.

For the authoritative discussion of CLOS MOP concepts, please refer to [Kiczales, Rivieres, and Bobrow].

3.1.3 Dynamic Object-oriented Programming

A chief truism for conventional database paradigms is that they do not allow dynamic augmentation of metadata. Due to the MOP's dynamic CLOS underpinnings, dynamically changing the value of shared facets, dynamically adding shared facets, and dynamically deleting shared facets becomes as simple as writing to a slot or redefining a class, respectively. The same is true for instance facets.

3.2 Choosing a Persistence Paradigm

Commercially supported, well developed, reliable CLOS persistence is rather new. Franz Inc., AllegroStore is a CLOS layer on top of ODI's ObjectStore, the number one selling object-oriented database [International Data Corporation]. Because ObjectStore has been proven by over five years of use, it strongly appears to be the best foundation upon which to build persistent facets. See Section 5.1 for actual implementational details concerning my usage of AllegroStore [Franz].

4 Comparison with Existing Work in AI and Databases

Slot facets are not a particularly new idea. Intellicorp's Knowledge Engineering Environment [KEE] employed shared facet functionality except that it did not have persistence. In fact, I borrow their term *facet* to name not only my similar shared metadata, but my instance metadata as well. [Mato Mira] and others have provided instance-facet-like functionality for CLOS, but without persistence. [Paepcke] extended CLOS via the MOP to add basic persistence to slot values, but did not address metadata. This toolkit attempts to maximize a CLOS look and feel, and I expect that many will find its use more intuitive than the other facet implementations.

5 Architectural High Points and Related Techniques Developed

5.1 Persistent Metaobjects

I subclassed AllegroStore's effective slot definition and mixed in their own persistence class into this subclass (see the definition of *omni-effective-slot-definition* in Figures 2 and 4). I then specialized the generic function *effective-slot-definition-class* to my metaclass so as to inform CLOS that it needs to use my specialized version of AllegroStore's effective slot definition. I then specialized CLOS's metaclass instance creation function so that it would try to get a slot definition from the database before trying to create one from scratch. This was the key to providing persistent shared metadata since the shared metadata *piggy-backs* on the effective slot definition objects.

Instance facets are made to persists in a similar yet simpler way.[5] Just as an application's class is made persistent by specifying its metaclass as *allegrostore:persistent-standard-class*, the same is true for instance facets. Since both persistent and non-persistent instance facets are likely required by an implementation, class mirroring of the type discussed in 5.4 is used as well. Implementation details of instance facets follow in Section 5.3.

[5] Since objects that implement an instance's slots very arguably are metaobjects, I will discuss their persistence in this section as well.

5.2 Metaclass Support for Metaobject Inheritance Orchestration

Because this tool suite's metaclasses are concocted mixtures of metaclass superclasses, complications in orchestrating the inheritance naturally arise (consult Figures 2, 3, and 4 as guides for understanding this section). As an example, *omni-class* is designed to contain all often used metaclass functionality.[6] Its metaclass must inherit from the instance facet metaclass which in turn inherits from the shared facet metaclass. This would be a simple matter of object-oriented inheritance, if a given metaclass did not often need other supporting metaobjects. For example, the shared facets functionality requires the use of extending subclasses of CLOS's slot definition objects. Additionally, the instance facet's effective and direct slot definitions also need to inherit from *shared-facets-effective-slot-definition* and *shared-facets-direct-slot-definition*, respectively. Other metaobjects which sometimes need to be included in this *parallel inheritance* are the generic function, method, reader, and writer metaobjects.[7] I provide a macro by the name of *defmetaclass* that knows how to find all key metaobject definitions relating to a given metaclass, and inherits all this related functionality from all metaclasses in the superclass list of the metaclass in question. More simply put, this is the parallel inheritance described in Figure 2.

5.3 Custom Mixed Sparse Instance Facets

In various implementations, instance facets have suffered from one of two maladies. Either they are (*i*) implemented as p-lists or some other such *sparse* representation and suffer from the associated slower access time, or they are (*ii*) implemented as objects, all of which contain each possible facet.[8] This, of course, uses excessive memory.

When a slot requests instance facets, the current implementation creates an object that has a slot for the *real slot value*, while it has other slots to contain instance facet values.[9] Because each *defclass* slot definition can request a different combination of the services offered by the metaclass, I chose to custom mix, at class compile time, individualized facet combinations. In short, each slot that requests instance facets has storage allocated for only those facets. As the number of instance facets increase, this could conceivably cause a combinatoric explosion of facet combination definitions, but I strongly expect that application programmers mostly will use a small set of facet combinations in their slots. Because these facet combinations are implemented as objects, facet access is the time required for a simple slot access rather than an O(n) lookup as in the p-list case. Thus, custom mixing achieves the best of both worlds. See Figure 5.

5.4 Persistent Class Mirroring

Early in the work, it became apparent that an application programmer would not want to support redundant class definitions for both persistent and non-persistent instances of the same class. AllegroStore does not provide a means to avoid the duplicate definitions at present. My solution is to automatically generate both types of definitions from one *defclass*-like macro. My expectation is that AllegroStore will soon provide a solution that involves the creation of just one class that is able to support both persistent and non-persistent instances.

6 Unique Aspects of this Work

The chief contributions of this work are persistent CLOS-based shared and instance facets. Since CLOS metaobjects are bona fide CLOS objects, the metaobjects that implement the shared and instance facets

[6] In this example it inherits from just two metaclasses. In reality it inherits from many other metaclasses whose functionality is not covered it this paper.

[7] Since Franz Inc. believes it expedient not to allow a class definition whose metaclass is *metafoo* to be in the same file as the definition of metaclass *metafoo*, I do not have *defmetaclass* generate quasi-*standard-object* subclasses whose metaclass is that of the *defmetaclass* form. For the time being, I generate these quasi-*standard-object*s by hand.

[8] I call p-lists sparse because only the facets used in the application exist and reside in the p-list. The opposite of sparse would be the case where an instance's facets are implemented as an object where these objects always contain all possible facets.

[9] The facet object currently resides in the respective slot of its parent object. When the MOP's Instance Structure Protocol dispute is settled and the result is implemented, facets will be mapped directly into the parent object thus removing a level of indirection. In this case, an object with one slot with one facet would actually have two slots. The facet's slot would be invisible to conventional slot access in the parent object.

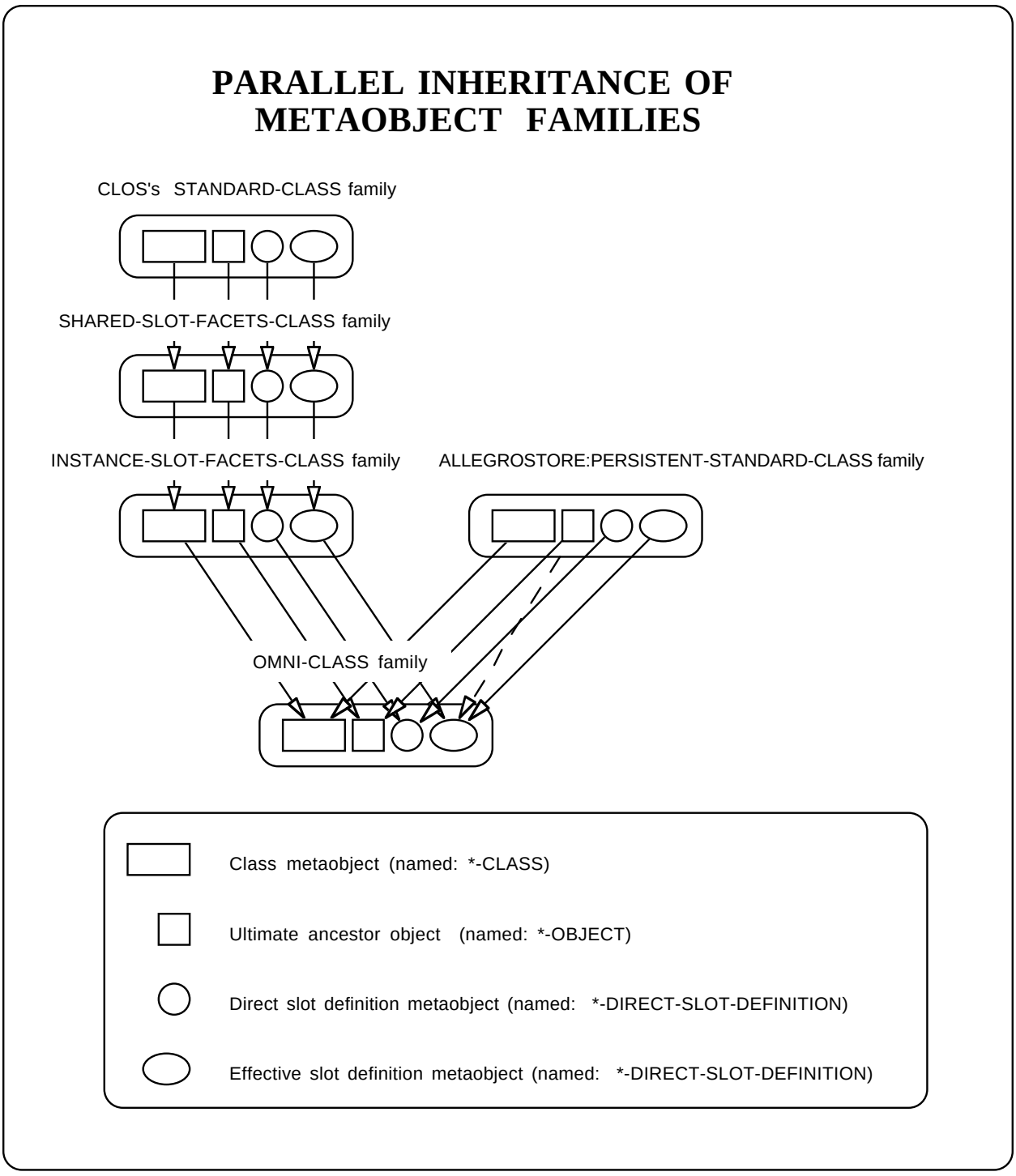

Figure 2: Arrows denote inheritance from the superclass to the subclass. Subclassing a metaclass often requires subclassing other metaobject classes in the metaclass's *family*. The parallel lines in this figure suggest the extra work involved in orchestrating *parallel* inheritance in systems consisting of complex hierarchies of metaclasses. See the next figure for the definitions of these families. Leaving the notion of parallel inheritance, please note that *omni-effective-slot-definition* gains persistence by inheriting from *allegrostore:standard-persistent-object*. This effective slot definition is of metaclass *allegrostore:standard-persistent-class*. This provides shared metadata persistence. Also be aware that the AllegroStore family does not follow the naming conventions suggested by the diagram.

```
(defmetaclass shared-slot-facets-class
  () ;STANDARD-CLASS is the default superclass for DEFMETACLASS
  () ;slots added to the metaclass would go here just as in DEFCLASS.
  (:documentation
   "This metaclass provides the ability to associate and retrieve
    information having to do with a particular slot of a given class."))



(defmetaclass instance-slot-facets-class (shared-slot-facets-class)
  ()
  (:slot-definitions-mixin-slots
   ;;the following slot will automatically get added in to both the
   ;;automatically generated
   ;;INSTANCE-SLOT-FACETS-EFFECTIVE-SLOT-DEFINITION
   ;;and INSTANCE-SLOT-FACETS-DIRECT-SLOT-DEFINITION metaobject
   ;;definitions
   (custom-mixed-slot-facet-class-object
     :accessor custom-mixed-slot-facet-class-object
     :initarg :custom-mixed-slot-facet-class-object
     :initform nil
     :documentation
     "This slot within the slot definitions stores the custom
      class definition of the object that will store all the
      instance facet information for in instance of metaclass
      INSTANCE-SLOT-FACETS-CLASS."))

  (:documentation "This metaclass provides the ability to associate
and retrieve information that belongs to a particular slot of a given
instance of a class."))



(defmetaclass omni-class
  (instance-slot-facets-class
   allegrostore:persistent-standard-class)
  ()

  (:effective-slot-definitions-mixin-slots
   ;;the following slot will automatically get added in only to the
   ;;automatically generated OMNI-EFFECTIVE-SLOT-DEFINITION
   (unique-perma-identifier
    :type symbol
    :accessor unique-perma-identifier
    :initarg :unique-perma-identifier
    :initform nil
    :allocation :persistent
    :documentation
    "used for helping to uniquely identify the persistent effect
     slot definition in memory at class creation time"))

  (:slot-definitions-mixin-slots
   (clos::allocation
    :initform :persistent
    :documentation
    "This causes slots to be persistent by default.  AllegroStore's
     default is non-persistent"))
  (:documentation
   "OMNI-CLASS is intended to be the generally used metaclass.
    It should contain all frequently used metaclass functionality."))
```

Figure 3: An instance of the *defmetaclass* macro defines not only the metaclass itself, but the metaclass's entire family of metaobject classes. The best known example of a metaobject class family is the *standard* metaobject classes described in [Kiczales, Rivieres, Bobrow]. I refer to them as the *standard-class* family. See the next figure.

```
(LET
 ((CLASS-METAOBJECT
       (DEFCLASS OMNI-CLASS
           (INSTANCE-SLOT-FACETS-CLASS
            ALLEGROSTORE:PERSISTENT-STANDARD-CLASS)
         NIL
         (:DOCUMENTATION
          "OMNI-CLASS is intended to be the generally
           used persistent metaclass for knowledge representation.
           It should contain all frequently used metaclasses."))))

     (CLOS:FINALIZE-INHERITANCE CLASS-METAOBJECT)

 (EVAL-WHEN
  (:COMPILE-TOPLEVEL :LOAD-TOPLEVEL :EXECUTE)

  (DEFMETHOD ULTIMATE-ANCESTOR-OBJECT-CLASS-GIVEN-METACLASS
      ((OMNI-CLASS
        (EQL 'OMNI-CLASS)))
    (FIND-CLASS 'OMNI-OBJECT)))

 (EVAL-WHEN
  (:COMPILE-TOPLEVEL :LOAD-TOPLEVEL :EXECUTE)

  (CLOS:FINALIZE-INHERITANCE
   (DEFCLASS OMNI-DIRECT-SLOT-DEFINITION
       (INSTANCE-SLOT-FACET-DIRECT-SLOT-DEFINITION
        ALLEGROSTORE::P-SC-DIRECT-SLOT-DEFINITION
        CLOS:STANDARD-DIRECT-SLOT-DEFINITION)
     ((CLOS::ALLOCATION
       :INITFORM :PERSISTENT))))

  (DEFMETHOD CLOS:DIRECT-SLOT-DEFINITION-CLASS
      ((OMNI-CLASS
        OMNI-CLASS)
       &REST INITARGS)
    (DECLARE (IGNORE INITARGS))
    (FIND-CLASS 'OMNI-DIRECT-SLOT-DEFINITION))

  (DEFCLASS OMNI-EFFECTIVE-SLOT-DEFINITION
      (INSTANCE-SLOT-FACETS-EFFECTIVE-SLOT-DEFINITION
       ALLEGROSTORE::P-SC-EFFECTIVE-SLOT-DEFINITION
       ALLEGROSTORE:PERSISTENT-STANDARD-OBJECT
       CLOS:STANDARD-EFFECTIVE-SLOT-DEFINITION)
    ((CLOS::ALLOCATION
      :INITFORM :PERSISTENT)
     (UNIQUE-PERMA-IDENTIFIER
      :TYPE SYMBOL
      :ACCESSOR UNIQUE-PERMA-IDENTIFIER
      :INITARG :UNIQUE-PERMA-IDENTIFIER
      :INITFORM NIL
      :ALLOCATION :PERSISTENT
      :DOCUMENTATION
      "used for helping to uniquely identify the persistent effective~
       slot definition in memory at class creation time"))
(:METACLASS ALLEGROSTORE:PERSISTENT-STANDARD-CLASS))

  (DEFMETHOD CLOS:EFFECTIVE-SLOT-DEFINITION-CLASS
      ((OMNI-CLASS
        OMNI-CLASS)
       &REST INITARGS)
    (DECLARE (IGNORE INITARGS))
    (FIND-CLASS 'OMNI-EFFECTIVE-SLOT-DEFINITION))))
```

Figure 4: This is a simplified version of the macro-expansion for the *defmetaclass* macro for *omni-class*. Note that the macro is able to determine the specific ancestry of each member of the *omni-class* metaclass family and thus orchestrate and simplify parallel inheritance.

```
;;;The following class definition has four different slots
;;;with four very different storage needs:
(defclass person (mammal)
  ((employer ;This slot requires storage to keep a record of its previous contents.
    :slot-state-history t)

   ;;This slot simply wants all PERSON instance
   ;;height slots to annotate from where the height
   ;;information came.
   (height
    :information-source t)

   (address ;This slot wants allocation storage for both facets.
    :slot-state-history t
    :information-source t)

   (age) ;This slot requests no facets and remains implemented as a vanilla CLOS slot.
   ... )
  (:metaclass omni-class))

;;;This is mixed into all instance facet containers
(defclass custom-mixed-slot-perma-facet-class ()
  ((slot-value-facet
    :accessor slot-value-facet
    :initarg :slot-value-facet
    :allocation  :persistent
    ))
  (:documentation
   "Used to store the value of the slot containing
    this facet object.")
  (:metaclass allegrostore:persistent-standard-class))

(defclass example-facet-container-for-person--employer
  (custom-mixed-slot-perma-facet-class)
  ((slot-state-history :type list :accessor slot-state-history
                       :initarg :slot-state-history :initform '()
                       :allocation :persistent))
  (:metaclass allegrostore:persistent-standard-class))

(defclass example-facet-container-for-person--height
  (custom-mixed-slot-perma-facet-class)
  ((slot-information-source
    :accessor slot-information-source
    :initarg :slot-information-source
    :allocation :persistent))
  (:metaclass allegrostore:persistent-standard-class))

(defclass example-facet-container-for-person--address
  (custom-mixed-slot-perma-facet-class)
  ((slot-state-history :type list :accessor slot-state-history
                       :initarg :slot-state-history :initform '()
                       :allocation :persistent)
   (slot-information-source
    :accessor slot-information-source
    :initarg :slot-information-source
    :allocation :persistent))
  (:metaclass allegrostore:persistent-standard-class))
```

Figure 5: Instance facets require extra storage for each facet of each instance's slot. Each of *person*'s slots in this example has a different combination of facets. For each unique combination of facets requested for a slot in a class definition, the system will automatically generate a unique facet container definition. Each *person* instance will have its appropriate slots contain instances of these custom facet containers.

can be made persistent via CLOS persistence packages such as AllegroStore.[10]

Additional original aspects include persistent metaobjects, parallel inheritance support, and custom mixed facets as described in Section 5.

7 Proposed Uses

The areas in which metadata might be useful seem utterly limitless. Metadata for digital media is one area attracting particular attention as seen in a recent issue of *SIGMOD Record* [Segev]. This issue was devoted entirely to metadata. Some particular uses for metadata highlighted in this issue were multi-media documents, video databases, image search, satellite images, browsing of structured media objects, speech documents, and mixed-media access. Refer to Sections 2.1 and 2.2 for listings of envisioned and present uses of shared and instance metadata, respectively.

8 Conclusions

The non-persistent version of this code has proved markedly helpful in two projects over the past year and a half. One example is that of shared facets. They proved to be a CLOS-like, intuitive means of storing database consistency constraints as described in Section 2.1. Slot update constraints were stored as shared facets and entered as CLOS-like slot options as in Figure 1. Since these slot update constraints were part of the *defclass* definition, no updating of constraint definitions was necessary when class or slot names were changed.

It is my hope that the persistent version of this tool suite will prove to be a foundation for the many in the Common Lisp community to build upon — providing both knowledge representation intensive applications as well as ordinary applications with a decisive increase in representational power.

The persistent aspects of this code are in an alpha-test state; the non-persistent aspects are in beta-test condition. Interested parties may contact me about obtaining the source code.

[10] This system used to support ITASCA Corporation's ITASCA persistence before ITASCA Corporation's demise. My support for ITASCA has since lapsed.

References

[1] Steele, G. *Common Lisp the Language (Second Edition)*, Digital Press, U.S., 1990.

[2] Seligman, L., *A Mediated Approach to Consistency Management in Distributed Heterogeneous Information Systems*, Ph.D. Thesis, Information Technology, George Mason University, Fairfax, Virginia, 1994.

[3] P. Maes, D. Nardi, *Meta-Level Architectures and Reflection*, page *vii*, North-Holland, Amsterdam, The Netherlands 1988.

[4] Kiczales, G., Rivieres, J., Bobrow, D., *The Art of the Metaobject Protocol*, The MIT Press, Cambridge, MA, 1993.

[5] International Data Corporation, *Object Technologies*, ref. IDC #8790, April 1994.

[6] Franz Inc., *AllegroStore Manual*, Version 1.0, USA, 1989.

[7] IntelliCorp Inc., *KEE Core Reference Manual*, USA, 1989.

[8] Mato Mira F., "ECLOS: An Extended CLOS" in *Object-Oriented Programming in Lisp: Languages and Applications Workshop ECOOP 93*, Kaiserslautern, Germany, July 1993.

[9] A. Paepcke, "PCLOS: A Flexible Implementation of CLOS Persistence" in *Proceedings of the European Conference on Object-Oriented Programming*, pages 374-389, 1988.

[10] A. Segev. (Ed.), *SIGMOD Record*, Vol. 23, No. 4, ACM Press, New York, NY, 1994.