ADHOC and the MAP

ADHOC

Another Declarative Hierarchical Object-centric CLOS Customization. An advanced object oriented extension to CLOS. The **defobject** macro is the interface of *ADHOC*. Before we get to the language description, lets show some examples:

This is a minimal **defobject** definition:

```
(defobject foo ())
==> #<ADHOC-CLASS RGN::FOO>
(make-instance *)
==> #<FOO {1026C32823}>
(setq self *)
(the superior)
==> NIL
(the root)
==> #<FOO {1026C32823}>
```

The macro **the** sends messages to the binding of the variable **self**. Normally you only bind **self** globally for debugging. As we can see the instance of **foo** has no **superior**. This is because it was instantiated the top level. But the instance does have a **root**, which is itself. If this doesn't make sense now, it will make sense when we talk about component attributes.

To send a message to any object, use the macro **send**. The macro **the** which is shorthand for (**send self** <message>*) is normally used inside the **defobject** macro.

Example:

```
(send self root)
==> #<FOO {1026C32823}>
```

Now lets define some attributes and see a more interesting result from **send**:

```
(defobject foo ()
:attributes ((one 1)
            (two (+ 1 (the one)))))
==> #<ADHOC-CLASS RGN::FOO>
```

one and **two** are names of attributes.

```
(setq f (make-instance 'foo))
```

```
    ==> #<FOO {101D366F53}>
    (send f two)
    ==> 2
```

As you may have surmised, inside the body of the attribute **two**, **self** is bound to the instance of **foo**. **(the one)** expands to **(send self one)** which then expands to **(slot-value self 'one)**. So *attributes* are CLOS slots. They are CLOS slots which are customized to compute, and in general, cache the result. They also keep the result from becoming stale.

Consider the following:

```
    (defobject foo ()
    :inputs
    ((one 1))
    :attributes
    ((two (1+ (the one)))))
      (setq f (make-instance 'foo))
      (send f two)
      ==> 2
      (setf (slot-value f 'one) 10)
      (send f two)
      ==> 11
```

The term *attributes* is overloaded. Generally in this document it means all *attribute classes* which specify types of computed slots, such as *inputs*, *components* and *attributes*. The type of attribute class simply called a*ttributes* refers to attributes classes which are neither *inputs* nor *components*. Specifically, these are *ordinary attributes*, *uncached attributes* and *modifiable attributes.* In this document it should be clear by the context whether *attributes* means *all* computed slot types or *ordinary attributes*, *uncached attributes* and *modifiable attributes*, but if it is not it will be made explicit.

There are three types of inputs. The one (excuse the pun) shown in the example above is an *optional input*. The are also *ordinary inputs* and *defaulting inputs.* As mentioned, there are three types of attributes. **two** in the example above is an *ordinary attribute*. Currently, there are three types of component attributes, *ordinary components* and two types of *aggregate components*: *array aggregates* and *table aggregates*. Component attributes return objects of type **object**, or its subclasses. **object** is the base ADHOC type.

But before components are discussed, consider inheritance:

```
(defobject bar (foo)
:inputs
((one 11)))
(setq b (make-instance 'bar))
(send b two)
==> 12
```

The class **bar** inherits the attribute **two** from the class **foo**. The class **bar** overrides, or *shadows*, the attribute **one** from the class **foo**.

An example of shadowing:

```
(defobject bar (foo)
:attributes
((one 11)))
(setq b (make-instance 'bar))
(send b two)
==> 12
(setf (slot-value b 'one) 12)
==> Error: slot ONE of type EFFECTIVE-ORDINARY-ATTRIBUTE-DEFINITION is read only
```

Oops. Notice how one can shadow one type of slot, such as an optional input, with another type of slot such as an ordinary attribute using inheritance.

As said previously, component attributes return objects when messaged. *Aggregate components*, as the name suggest, return one of multiple objects, you must pass the *indices* of the particular object to **send**, in the form of the list of the message followed by the indices, such as (**send self (parts 7)**). Generally you will only have one index.

First an example of an *ordinary component*:

```
(defobject assembly ()
:inputs ((component-class (find-class 'foo)))

:components
((part-1 :type (the component-class)
   one 3
   two 3
   three 3)))
(setq assy (make-instance 'assembly))
(send assy part-1 one)
```

```
==> 3
(send assy part-1 two)
==> 4
(send assy part-1 three)
==> 3
```

A component attribute definition is a list, followed by the name of the component attribute, i.e. **part-1**. Next is the **:type** keyword, followed by a form which evaluates to a class or a class-name. Finally the rest are initarg/expression pairs where the initarg is the input name and the expression is what should be evaluated when the input of the component instance is messaged. But why is **(send assy part-1 two)** returning **4** and not **3**? Because **two** was last defined as an *ordinary attribute*, which is an *output* not an *input*! Also, what's up with **three**? **three** is what is called a *pseudo-input*, it's answered even though there is no **three** input defined for the **foo** class, but it's not cached.

Notice how **send** is taking three arguments now. This is called reference chaining, and it's equivalent to:

```
(slot-value (slot-value assy 'part-1) 'three)
```

Reference-chaining helps make message sending more readable. It is actually a lot like C++, with the **.** (dot), or **->** syntax.

In case you are curious:

```
(send assy part-1)
==> #<FOO {103885EED3}>
(send assy part-1 superior)
==> #<ASSEMBLY {1036F4F3B3}>
```

- #<FOO {103885EED3}> (aka **part-1**) has a superior now. **part-1** is considered a child, or component of #<ASSEMBLY {1036F4F3B3}>.

Aggregate component definitions are the same as ordinary component definitions, with the exception of the **:aggregate** keyword/value pair. **:aggregate** is followed by a list starting with either **:size** or **:indices** followed by an expression which computes the size of the array or the indices of the table. The macro **the-part** expands to **(send part** <message>*), where **part** is bound to the particular component in question.

Example:

```
(defobject assembly ()
:inputs ((component-class (find-class 'foo)))

:components
((parts :type (the component-class)
         :aggregate (:size 10)
        one (the-part index)
        two 3
        three 3)))
(setq assy (make-instance 'assembly))
(send assy (parts 1) one)
==> 1
(send assy (parts 2) two)
==> 4
(send assy (parts 5) three)
==> 3
```

This has been a simple introduction to the pure programming-language features of *ADHOC*. So far, without being overly technical, we have shown a simple language where objects have attributes, which are inputs and outputs, and inferiors (children) where attributes can be computed and referenced, and inferior objects can be generated and customized. ADHOC classes can be used just like ordinary classes in CLOS for use in *generic function method specializers*. ADHOC simply extends CLOS with a hierarchical object-centric message passing computed attribute paradigm.

Now a language description of ADHOC:

Below is a BNF chart of the syntax of **defobject**. It provides reference for most of the functionality that ADHOC objects can have. Other than that, ADHOC classes and instances are just normal CLOS classes and instances.

```
(defobject <name> (<superclass>*) <generalized-attribute-section>*)
  <name> ::= a lisp symbol for the name of this class
  <superclass> ::= a lisp symbol which specifies a class to inherit from
  <generalized-attribute-section> ::=
    <inputs-section> ::=
      :inputs (<input-specification>*)
         <input-specification> ::= <attribute-name> | (<attribute-name> <input-declaration>*
                                                        <form>* <attribute-expression>?)
            <attribute-name> ::= a lisp symbol
            <attribute-expression> ::= a lisp expression which evaluates to any lisp object, and
                                       will be used as the initial value of the input if
                                       not provided
      <input-declaration> ::= :defaulting | :descending | <noticer-form>
      <noticer-form> ::= (:noticer <form>* <noticer-expression>)
        <noticer-expression> ::= a lisp expression to be executed where self is bound to
                                    the instance and value is bound to the value the slot
                                    is being set to
        <form> ::= a lisp expression, evaluated for side effect
<attributes-section> ::=
  :attributes (<attribute-specification>*)
    <attribute-specification> ::= (<attribute-name> <attribute-declaration>*
                                    <form>* <attribute-expression>)
      <attribute-declaration> ::= :uncached | :modifiable |
                                    <noticer-form> | :descending
        note: :noticer is only compatible with :modifiable in
         the :attributes section.
       :modifiable and :uncached are mutually exclusive
<components-section> ::=
  :components (<component-specification>*)
    <component-specification> ::= (<attribute-name> :type <class-expr>
                                       <keyword-expression-pair>*) |
            (<attribute-name> :type <class-expr>
                                      :aggregate (<aggregate-spec-keyword> <form>)
                                           <keyword-expression-pair>*)
      <class-expr> ::= lisp expression resolving to a class-name or class-object
    <aggregate-spec-keyword> ::= :size | :indices
      <keyword-expression-pair> ::= <input-keyword> <input-expression>
        <input-keyword> ::= a slot name of a child input
        <input-expression> ::= a lisp expression when executed computes the value for
                                   the child input, this expression overrides the optional
                                   input expression in the child's defobject.  Note: the
                                   symbol self is bound to the parent and the symbol
          part is bound to the child
<slots-section> ::=
  :slots (<slot-specification>*)
    <slot-specification> ::= an ordinary CLOS defclass slot specification
<metaclass-section> ::= :metaclass <class-name>
```

The metaclass section can only appear once in a **defobject** form. The metaclass must be a subclass of **adhoc-class**.

Meta-Attribute Protocol

The MAP is a suggested list of attribute classes that can be implemented in CLOS to make Object Oriented Programming even more powerful and convenient. Attributes are computed slots. The MAP can be conveniently implemented in the Meta-Object Protocol (MOP) as ADHOC is. The MOP is a de facto standard established by the book, the Art of the Metaobject Protocol, which specifies the interface for extending the Common Lisp Object System. Not all CLOS implementations support the MOP. The MAP is protocol which defines concepts which were initially part of the Icad Design Language which predates CLOS. CLOS is the object system which was decided upon as the standard for Common Lisp. Its predecessors include Flavors, which was the implementation object system for the Icad Design Language. Another example of MAP-like functionality includes Genworks' Gendl. Some concepts, such as message passing, which is a feature of Icad, Gendl, and ADHOC come from Flavors which in turn inherited that feature from Smalltalk.

Terminology

The terms parent/child, assembly/component, superior/inferior, superior/subordinate, assembly/part, whole/part, and container/contents all refer to the same concept. Essentially, one object is an attribute value of another object. These terms may be used interchangeably depending on what word is most appropriate. The parent/child relationship is different from the superclass/subclass hierarchical relationship. The superclass/subclass relationship is a type hierarchy not a component relationship. For example, a car is a vehicle vs a car has an engine. Both types of relationships have modes of inheritance however.

Dependency management refers to how attributes get updated when a slot which the attribute form of another slot is dependent on gets **setf**'d. An attribute form is a chunk of Lisp code which returns the value of the slot. Not all attributes or slots have an attribute form.

Attribute Classes

(aka meta-attributes)

Ordinary CLOS Slots

Ordinary CLOS Slots are basically named storage locations in a CLOS instance. See a textbook or tutorial on CLOS to find out more about CLOS slots. They do not compute a value. **slot-value** will only return the current value at the storage location or an unbound slot error. **(setf slot-value)** will set the value of the slot.

Settable Slots

This is a supertype of Meta-Attributes which allow the attribute value to be **setf**'d. The Meta-Attributes which are settable include all the input types, and the modifiable attribute type. Setting a slot may trigger a noticer or dependency management action. Ordinary CLOS slots are implicitly settable but do not inherit from this supertype.

Non-Settable Slots

This is a supertype of all other meta-attributes besides inputs and modifiable attributes. Attempting to **setf** such a slot will result in an error.

Ordinary Inputs

Inputs are attributes which can be set at any time, including object instantiation time, or respond to 'provided' inputs given by the parent to the child in the component definition. Ordinary Inputs do not have an initial value unless one is provided to **make-instance**.

Optional Inputs

Optional Inputs are like ordinary inputs except they provide a form which can be evaluated at messaging time to compute the initial value.

Once an input is externally set, by **(setf slot-value)**, the value will stay and not be disturbed by dependency management. Computed values can be invalidated, however. In this case, if an optional input is dependent upon another attribute which changes, the input's attribute form will be re-evaluated upon messaging.

Defaulting Inputs

Defaulting inputs are inputs which will search the list of ancestors for a value from an attribute of the same name to answer the message.

Defaulting Ordinary Inputs

Defaulting ordinary inputs will error with **slot-unbound** if no ancestor is found to answer the message.

Defaulting Optional Inputs

Defaulting optional inputs provide a form to be evaluated if no ancestor is found to answer the message. Similar rules apply to defaulting optional inputs as to optional inputs with regards to recomputation.

Ordinary Attributes

An ordinary attribute is a non-settable slot which provides a form to be evaluated to compute/recompute the value of the slot.

Uncached Attributes

An Uncached attribute is a slot in spirit but has no storage location. The form is evaluated every time it is messaged.

Modifiable Attributes

Modifiable attributes act like state variables, as they are settable slots but do not accept input. Like an optional input the attribute form will be evaluated to provide a default value, but once the slot is **setf**'d the value stays until **setf**'d again.

<u>Components</u>

<u>Ordinary Components</u>

An ordinary component is a child, or **part** of the encapsulating class. It takes a **:type** input whose form either evaluates to a class object or a class name of the child being instantiated and takes a parameter list of input names and input forms to provide behavior to the subordinate. This is key to the generative capabilities of ADHOC with it's abilities to override default input forms.

<u>Aggregate Components</u>

Aggregate components are arrays and tables of objects with indices. The type form either evaluates to a class object or class name for the parts or a sequence of class objects or class names for the parts. The provided inputs are the same as with ordinary attributes, but can conditionalize on **indices** as well.

<u>Descending Attributes</u>

Descending attributes are not an attribute class, rather they are any attribute with the **:descending** property. The **:descending** property allows descendants of the object to answer the message. Component attribute names are always descending.

<u>Part/Whole Defaulting</u>

The property of component attribute names always being descending attributes.

<u>Noticers</u>

Noticers are functions provided to a settable slot which is called on **self** and **value** when the slot is set. There can be multiple noticers on a settable slot and they run in the order which they are defined. Due to the fact that attributes of a particular name shadow attribute of that particular name from the superclass, noticers are are not inherited when shadowed, only when not shadowed by a subclass. In other words, noticers belong to the specific attribute definition where they are defined.

<u>Built-in Attributes</u>

Inherited ordinary clos slots for every object are **root, superior, component-definition, aggregate,** and **indices**. Computed slots inherited by every object: **children,** and **root-path**.

- **root** - contains the top-level object in the parent/child tree. Can be **nil**.
- **superior** - contains the parent of an object. Can be **nil**.
- **component-definition** - an object with metadata, similar to a class, which partially describes the behavior of the object. Can be **nil**.

- **aggregate** - if the object is part of an aggregate component, this is the object that provides the array or tables. Can be **nil**.
- **indices** - if the object is part of an aggregate component, this is its set of indexes.
- **children** - all the children/parts/components of this object
- **root-path** - the reference chain of messages to go from the root object to this object.

Slots for the aggregate object of an aggregate part include **root, superior, component-definition, list-elements** and **size** for array aggregates, and **indices** for table aggregates.