Type-Definition Objects

William H. Duquette
Jet Propulsion Laboratory, California Institute of Technology
William.H.Duquette@jpl.nasa.gov

ABSTRACT

The dominant Tcl object paradigm is "objects are ensemble commands". However, the call-by-name semantics associated with this paradigm are not always appropriate. Type-definition objects provide a simple way to combine complex object behavior with call-by-value semantics. A type-definition object is an ensemble command which defines the operations available for a type whose values are represented as standard Tcl values. Operations include creating new values, modifying values, querying values, conversion of values from other types, validation of values, and so forth. A type-definition object can be a singleton, as in a "matrix" command which defines a variety of operations on general matrices represented as lists-of-lists; the Tcl 8.5 "dict" command and the family of "list" commands can be regarded as examples of type definition singletons. Type-definition objects can be also be parameterized instances of a class of type definitions; objects of this kind can define ranged numeric types, enumerations, and other constrained types. This paper presents a number of type definition objects implemented as part of the Joint Asymmetric Warfare Simulation (JAWS).

1. VALUE AND OBJECT SEMANTICS

It is commonly said that in Tcl, "everything is a string." And it is true that in a Tcl program, every entity has a string representation of some kind—but not all string representations convey the same amount of information. Some strings have value semantics, while others have object semantics.

A Tcl list is a classic example of an entity with value semantics. A Tcl list is a data value. Represented efficiently as an array internally, a Tcl list can be converted to and from its string representation without loss of information. It can be stored in a variable, passed from one procedure to another, written to a channel and read back again, and when it is no longer needed Tcl releases the memory automatically. Moreover, it is a chameleon: one and the same value can be a list, a string, a dictionary, a command to be evaluated, a stack to be pushed or popped, a record of data, a vector, or any number of other abstractions.

Most Tcl values share this ability to shift whimsically from one data type to another. Consider the value "99". It might be a two-character string, it might be a decimal integer, it might be a hexadecimal integer, it might be an index into a hash-table, or it might be a handle to some opaque data structure implemented in C code. It can be any or all of these things, and it can switch effortlessly from one use to another in successive lines of code, or even in the same line of code.

The one thing Tcl values lack is any kind of intrinsic behavior. Tcl values are not objects; they do not have methods. You can treat any Tcl value as a string; to go beyond that, you need to know something about the nature and provenance of the value.

Finally, Tcl values are transparent. Because every Tcl value has a canonical string representation, you can always see precisely what the value is.

Objects, by contrast, combine data and behavior into a single entity. The dominant object paradigm in Tcl is that pioneered by the Tk widgets: an object is represented as a Tcl ensemble command, that is, a command whose first argument is a subcommand and whose remaining arguments vary by subcommand.

An object's string representation is simply its name, which functions much like a pointer in conventional languages. The object name is a Tcl value which can be stored in a variable, passed to procedures, printed out and read in again, and so forth, just like any other Tcl value; yet, like a pointer it gives access to the object's behavior and associated data. Such pointer-like semantics are extremely useful, but in terms of data representation they come at a price: objects are opaque. Their data is stored inside, in either C data structures (for objects like Tk widgets and images) or hidden Tcl variables (for objects implemented using Snit[1] or some other pure-Tcl object framework). A program must use the object's methods to access the internal data—or, rather, that subset of it which is externally accessible. Thus, printing out an object or saving it to disk—that is, serializing it—is a tricky proposition for which there is no one-size-fits-all solution. Indeed, not all objects can be serialized in any meaningful way.

Another disadvantage of objects is the need for explicit memory management. Unlike a Tcl value, whose memory is released automatically when the value is no longer needed, a Tcl object must be explicitly destroyed. Further, with explicit memory management comes ownership management: the programmer must always keep in mind which module owns the object at each moment in its lifetime. Code that manipulates values can copy them and modify them as it pleases; code that manipulates objects must always be written with an awareness of which module owns the object and has the right to destroy it. As a result, Tcl objects (however implemented) are a poor choice for representing lightweight, short-lived data entities with value semantics.

How, then, should we associate behavior with our data values? The classic OO approach is to encapsulate the value within an instance of a class; but this is often unpleasantly heavy-weight in Tcl. We are left with a dilemma: values have transparency and value semantics, but have no behavior; objects have behavior, but lack transparency and value semantics and add increased management overhead.

The solution is to take a step backward from OO, and separate the code from the data.

2. TYPE DEFINITION SINGLETONS

The classic Tcl approach to combining behavior with value semantics is the procedurally-based abstract data type. The canonical example is the family of Tcl list commands: **list**, **lindex**, **lappend**, and so forth. This approach is now frowned on, at least in the Tcl core, because it pollutes the global namespace with an ever expanding number of new commands.

The modern approach is the *type-definition ensemble*, of which perhaps the best example is Tcl 8.5's new **dict** command.[2] This approach is functionally equivalent to the procedural approach, but the myriad of individual operations are implemented as subcommands of a single ensemble command, rather than as individual procedures: thus, **dict append** adds keys and values to a dictionary, **dict get** retrieves a value given its key, **dict set** replaces a key's value with a new value, and so on.

The advantages of this approach are three-fold. First, additional subcommands can be added as needed without any chance of name collisions with application-defined procedures. Second, there is only one command name to remember, which simplifies looking things up in the documentation. This advantage is often overlooked, but it should not be scorned: because of Tk's reliance on ensembles (and, in some cases, sub-ensembles), the Tk documentation is simultaneously concise, dense, and easy to use. Third, the commands needed to manipulate values of the type are packed into a single entity which has object semantics: it is referred to by a single name, which can be passed from procedure to procedure, stored in variables, and so forth. Because of this last property, a type-definition ensemble can be thought of as an object: a *type-definition singleton*.

2.1 Implementing Type-Definition Ensembles

Implementing a type-definition ensemble is a bit of a chore, simply because implementing well-behaved ensemble commands in pure Tcl is a bit of a chore.[3] Until recently, the easiest approach has been to note that any ensemble command can be regarded as a singleton object. Hence, one can define a class with the required subcommands as its methods, and define a single instance of the class; the instance is your ensemble. Alternatively, a Snit type with no instances makes a decent ensemble; here, for example, is a complete implementation of Tcl 8.5's **dict** command using Snit (the pragmas ensure that the resulting ensemble has no extraneous subcommands):

```
snit::type mydict {
    pragma -hasinstances 0
    pragma -hastypeinfo 0
    pragma -hastypedestroy 0
    delegate typemethod * using "::dict %m"
}
```

mydict is now a drop-in replacement for **dict**. Of course, it will only work properly in Tcl 8.5 and later....

Tcl 8.5's new **namespace ensemble**[4] command will be extremely helpful in this regard; with one line of code, a procedural interface defined in a namespace can be magically turned into an efficient ensemble command with decent error messages.

3. TYPE-DEFINITION CLASSES

Implementing a type-definition ensemble as a singleton object has some interesting implications. Suppose you want to define a bounded numeric type—probabilities, say, which are bounded between 0.0 and 1.0. You could define a Snit type:

```
snit::type probability {
    variable p 0.0

    method set {value } {
        if {$value < 0.0 || $value > 1.0 } {
            error "out of range"
        }

        set p $value
    }
    method get {} { return $p }
}
```

You *could* do that, but you'd be sorry, for all the reasons described in Section 1; for example, you can't use a probability represented in this way in an **expr** expression as easily as you can a probability represented as a floating point value. Not only does using a floating point value simplify your code, you'll generally know when you're dealing with a value that's a probability; there will be times when you'll need to verify that a putative probability value is a numeric value within the proper range, but you shouldn't need to do it every time you save a newly computed probability value.

Consequently, you'd much prefer to represent probabilities as normal Tcl values, which implies you might need a type-definition ensemble for probability specific behavior. Such an ensemble would have a method for verifying that a probability is both numeric and in range:

```
snit::type probability {
    typemethod inrange {p } {
        if {![string is numeric $p]} {
            return 0
        }

        expr {0.0 <= $p && $p <= 1.0}
    }

    ...
}

if {[probability inrange $p]} {
    ...
}
```

We now have a singleton object which verifies that numeric values are between zero and one. But suppose that there are a number of bounded numeric types of interest; we could define a type called **bounded** whose instances are parameterized with the desired minimum and maximum values. We would then have the ability to easily create a new type-definition ensemble for any range that takes our fancy:

```
bounded probability -min 0.0 -max 1.0
bounded percentage  -min 0   -max 100

set prob 0.5
set pct  50

if {[percentage inrange $pct]} {
    ...
}

if {[probability inrange $prob]} {
    ...
}
```

bounded may be termed a *type-definition class*. Instances of type **bounded** are *type-definition objects*. In a sense, they give us the best of both worlds. The types defined by **bounded** are normal Tcl values, with all of the advantages of value semantics; but all of the knowledge of how to use and manipulate values belonging to the the type is neatly encapsulated in an object which has object semantics. The type name (i.e., the name of the type-definition object) can be stored in a variable, passed to procedures, and used to parameterize other objects; or it can simply be defined as a global command, thus extending the range of types accessible to the entire application.

Moreover, because all instances of the type-definition class have the same operations, values of the defined types can be handled generically. Consider a table of named values which is to be read from a file:

```
prob1   0.5
prob2   0.3
pct1    57
pct2    23
```

The code to read and validate this table can be made completely generic simply by specifying a schema dictionary:

```
set schema {
    prob1  probability
    prob2  probability
    pct1   percentage
    pct2   percentage
}
```

This schema identifies the names of all valid table entries, and the type of each, with maximum clarity.

4. EXAMPLES

This section describes a number of type-definition singletons and classes used in the Joint Asymmetric Warfare Simulation (JAWS). The implementation is not shown (and indeed, it's rarely all that complex); of more interest is the style of programming which they enable.

4.1 Matrices

At the core of JAWS is a mathematical model of factions within a civilian population; it involves the relationships between the factions, as well as each faction's satisfaction level with respect to a number of concerns. The model involves a variety of matrices. The computational requirements are modest enough that there's no need to implement the matrix code in C, or use one of the C-coded numerical extensions; on the other hand, there's no reason to waste cycles either. Tcl lists are implemented internally as C arrays; a matrix implemented as a list of lists and accessed with **lindex** and **lset** is more efficient than a matrix implemented as a Tcl array with indices like "$i,$j".

JAWS defines a type-definition singleton, **mat**, for creating and initializing such matrices; it also provides a number of matrix computations and output operations. Here's a subset of **mat**'s subcommands:

mat new *m n ?initval?*
: Creates a new matrix of *m* rows and *n* columns whose elements are initialized to *initval*, which itself defaults to the empty string.

mat rows *matrix*
mat cols *matrix*
: Returns the number of rows and columns in the *matrix*.

mat add *matrix1 matrix2*
: Returns a matrix that's the sum of *matrix1* and *matrix2*.

mat format *matrix fmtstring*
: Applies **format *fmtstring*** to each element of *matrix* and returns the resultant matrix.

mat pprint *matrix rlabels clabels*
mat pprintf *matrix rlabels clabels fmt*
mat pprintq *matrix rlabels clabels quality*
: Each of these operations returns a text string containing the pretty-printed contents of the matrix. *rlabels* is a list of row labels and *clabels* is a list of column labels; *fmt* is a **format** string used to format each entry, and *quality* is the name of a **quality** object, of which more below.

mat doesn't define subcommands for setting and retrieving matrix elements; **lindex** and **lset** work perfectly well, and although **mat** could define subcommands that call **lindex** and **lset** there's no reason to pay the performance penalty.

4.2 Vectors

JAWS defines a type-definition ensemble, **vec,** for manipulating vectors. A vector value is simply a Tcl list; like **mat**, **vec** doesn't define its own version of the basic list operations, but instead defines operations specific to numeric vectors. The supported capabilities are analogous to those of **mat**; in addition, **mat** has subcommands which pull vectors out of matrices.

4.3 Enumerations

An enumeration is a mapping from symbolic constants to integers. Often a set of distinct constants is all that's required, and the mapping to integers is unimportant; Tcl programmers usually use a set of strings in such cases, and never define any explicit mapping to integers. In other cases, though, the mapping to integers is vital.

JAWS uses enumerations to define the rows and columns of the matrices in its mathematical model: each row index and each

column index is associated with a specific symbolic constant. For efficiency, numerical code needs to identify the rows and columns using numeric indices; for input and output, the mapping from symbols to integers and back again needs to be readily available. Consequently, JAWS defines a type-definition class called **enumeration**; instances of the class are type-definition objects which define specific enumerations. For example, JAWS models different factions in the civilian population; suppose there are three factions, A, B, and C:

```
enumeration faction {
    A  "Faction A"
    B  "Faction B"
    C  "Faction C"
}
```

The resulting type-definition object is called **faction**; note that each value in the enumeration has two names, a short name ("A"), and a long name ("Faction A"). The range of values of the type defined by **faction** consists of the long and short names listed above and the numeric indices 0, 1, and 2. The following are some of **faction**'s subcommands:

faction index *value*
faction shortname *value*
faction longname *value*
Returns the numeric index, short name, or long name associated with the *value*, which may have any of the forms listed above: short name, long name, or numeric index. If the *value* isn't valid for the type, **faction** throws an error.

faction size
Returns the number of distinct values in the enumeration, i.e., 3.

faction shortnames
Returns a list of the type's short names.

faction longnames
Returns a list of the type's long names.

In short, all of the knowledge about this enumerated type is encapsulated in a single named object which can be passed as an argument.

The faction names don't figure into JAWS' mathematical model, but the model code still needs to know them to support input and output. JAWS's computational engine is implemented as an object type; because enumerations can be represented by name, each instance of the engine can easily be given a different set of factions.

Note that the set of factions is not hardcoded, but is read from an input file; there's no reason why enumerations like **faction** need to have known definitions at implementation time.

4.4 Qualities

One of the inputs to JAWS is an initial value for the satisfaction of each faction with respect to a number of concerns; a satisfaction value is a number ranging from 100.0 to -100.0. To make populating the database easier, we provide a rating scale:

VS	Very Satisfied	80.0
S	Satisfied	40.0
A	Ambivalent	0.0
D	Dissatisfied	-40.0
VD	Very Dissatisfied	-80.0

On input, the user can enter a symbolic constant, which will be converted to the relevant number, or a specific number within the range 100.0 to -100.0, which we can either accept as is or replace with one of the numbers listed above. Further, given a numeric satisfaction value, we often wish to relate it to the nearest symbolic constant for output.

JAWS uses a total of nine different rating scales of this kind; each scale is defined by an instance of the **quality** type. For example,

```
quality qsat {
    VS "Very Satisfied"     80.0
    S  "Satisfied"          40.0
    A  "Ambivalent"          0.0
    D  "Dissatisfied"      -40.0
    VD "Very Dissatisfied" -80.0
} -min -100.0 -max 100.0 -format "%6.3f"
```

Unsurprisingly, this is similar to an **enumeration** definition; it adds a column for the numeric equivalents for each symbolic constant, and also has several options: the valid range for numeric satisfaction values is given, and a **format** string defines the standard output format and precision for values of the **quality** type.

A **quality** object like **qsat** has all of the subcommands that an **enumeration** does, i.e., **index**, **shortname**, **longname**, **size**, **shortnames**, **longnames**; in addition, it has the following subcommands:

qsat format *value*
Formats a numeric satisfaction value using the **–format** format string.

qsat value *value*
Returns a numeric satisfaction value corresponding to *value*. If *value* is a long or short name, the associated numeric value is returned. If *value* is already numeric, and is within the valid range for the type, it is returned unchanged. An error is thrown for any other input.

qsat strictvalue *value*
This is equivalent to **qsat value**, except that numeric inputs are mapped to one of the specific numeric values specified in the type definition.

qsat inrange *value*
Returns true if *value* is numeric and in the valid range, and false otherwise.

qsat clamp *value*
Clamps the *value* within the valid range, i.e. if *value* is 101.0 then 100.0 is returned. Numbers within the valid range are returned unchanged.

4.5 Interactions Between Types

These type-definition objects pack a considerable amount of knowledge into a small package, and as such are useful all on their own. However, the real payoff comes when they are used in tandem.

For example, JAWS defines a large number of matrices whose indices are defined by enumerations and whose values are defined by qualities. A satisfaction matrix, for instance, has rows indexed by a **concern** enumeration, columns indexed by a **faction** enumeration, and elements defined by the **qsat** quality. The **mat** type-definition ensemble defines two operations which support this pattern directly.

The first is **mat numerize**. Given a matrix of quality values, which may be symbols or numbers, and the name of the quality, **mat numerize** converts all matrix elements into numbers, and along the way validates them with respect to the quality:

```
% set a {
    {VS VD }
    {D  A  }
}
% set b [mat numerize $a qsat]
{80.0 -80.0} {-40.0 0.0}
%
```

The second is **mat pprintq**. This operation pretty-prints a matrix just as **mat pprint** does, but uses a quality object to include both symbolic and numeric values for each element.

```
% mat pprintq $b \
    [concern shortnames] \
    [faction shortnames] \
    qsat
              A           B
CON1  VS= 80.000  VD=-80.000
CON2   D=-40.000   A=  0.000
%
```

5. TRANSPARENT OO FOR TCL

Neil Madden has been pioneering "Transparent OO for Tcl,"[5] which indeed succeeds in adding object-like behavior to transparent data values. A transparent object, or "TOOT" is a list whose first element is a command that implements the object's behavior. Tcl's **unknown** command is extended to handle such lists in a special way: when a TOOT appears as the first token in a command, **unknown** calls the command whose name is embedded in the TOOT, passing it the TOOT itself followed by any additional arguments. The TOOT's command then has access to the TOOT's data, and can dispatch methods based on the next argument in the usual way.

This is an ingenious scheme, but it has two drawbacks, one of them fatal. The first is simply that TOOT values lose that chameleon-like flexibility, simply because the TOOT type's command name is embedded in the TOOT value: "90210" might be a string, an integer, or a ZIP code, but "::Zip | 90210" can only be a ZIP code.

The fatal drawback, however, is runtime efficiency; in practice, **unknown**-based dispatch is horribly slow, which makes it unsuitable for precisely those lightweight uses for which traditional objects are unsuitable. Nevertheless, the technique is worth remembering; and it's possible that future versions of Tcl will provide faster mechanisms for TOOT method dispatch. In the meantime, the techniques described in this paper for defining behavior for transparent Tcl values are superior in practice.

6. CONCLUSIONS

Type-definition ensembles are an established Tcl idiom for encapsulating behavior for Tcl data types with value semantics. Given an object system, type-definition classes and type-definition objects are a natural way to create families of type-definition ensembles. Because type-definition objects encapsulate knowledge about a type within a single command, they can be used to write generic algorithms which will apply to values of any type within a well-defined family. Because they can be defined at run-time based on application input, they are a convenient mechanism for describing, validating, manipulating, and formatting values based on user-defined schemas. And given the availability of multiple choices of object framework in the major Tcl distributions, there's no reason not to use them.

7. REFERENCES

[1] Duquette, William H., "Snit's Not Incr Tcl", http://www.wjduquette.com/snit.

[2] Tcl 8.5 Manual, http://www.tcl.tk/man/tcl8.5/TclCmd/dict.htm.

[3] Duquette, William H., "Guide to Creating Object Commands", http://www.wjduquette.com/tcl/objects.html.

[4] Tcl 8.5 Manual, http://www.tcl.tk/man/tcl8.5/TclCmd/namespace.htm.

[5] Madden, Neil E., "Transparent OO For Tcl", http://wiki.tcl.tk/11543.

8. ACKNOWLEDGEMENTS

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, during the development of the Joint Asymmetric Warfare Simulation (JAWS) for the U.S. Army's National Simulation Center at Fort Leavenworth, Kansas.