Tcl Beans:

Persistent Storage of TclOO Objects With Minimal Implementation Overhead

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

Abstract

A Tcl bean is an instance of a TclOO class representing a simulation entity with associated data. The set of all beans can be saved and later restored. Beans can own other beans, and the bean infrastructure supports bean deletion with undo, including cascading deletion, and bean copy and paste. The paper describes the bean implementation, including the current constraints on bean classes, as well as lessons learned while coming to grips with the TclOO framework.

1. Background

The Athena Regional Stability Simulation is a model of political actors and the effects of their actions in a particular region of the world. The region is divided into neighborhoods, in which reside various civilian groups. The actors have a variety of assets, including money, military and police forces, and means of communication, which they use to achieve their political ends. The extent to which they succeed depends on the attitudes of the civilians, which change in response to current events. The set of tactics an actor uses to achieve his goals is called his strategy.

1.1 The Problem

The Athena simulation model contains many different kinds of simulation entity. Most of these entities are implemented as a simple record structure (i.e., a dictionary) coupled with a type-definition ensemble [1]. Athena is a document-centric application, and to ease saving and restoring the scenario data most of the simulation entities are stored at run-time in an SQLite3 database called the *run-time database* (RDB). Each entity type gets a table; each entity gets a row in the table, and each record field gets a column. Entities are extracted from the RDB as required. This pattern has a number of benefits in addition to simplifying the data persistence problem: many of the entities serve as indices in arrays, which are conveniently represented as RDB tables, and many of the algorithms work well as batch operations on tables. Further, the ability to use SQL queries as the basis for algorithms and for data output has been a real win.

Over the last few years, however, we've added an increasing number of entities for which this pattern is at best unnatural. These entities are:

- Polymorphic: any given entity type has many subtypes
- Linked into complex tree structures
- Always processed one at a time, rather than in bulk

The first two points are notoriously inconvenient to deal with in SQL, and the third point means that there's no real advantage to dealing with the inconvenience. Consequently, we've been looking for an alternative architecture for this data.

1.2 The Solution

Polymorphism naturally suggests an object-oriented solution with inheritance, so we began to experiment with a TclOO-based solution. On the face of it, this is both straightforward and appealing: our entity types and subtypes become classes and subclasses, and entities are simply instances of classes. We get inheritance, encapsulation, and efficient execution, and of course objects can be easily linked into complex trees or networks in any of a number of ways.

However, we still had to support persistence. In the past, Athena data has been persistent in one of two ways: either it resides in the RDB to begin with, or it is stored in array variables owned by some singleton. The singleton can produce a checkpoint of its data, a nested dictionary that can be saved in the RDB as a string and can easily be pushed back into its arrays on restore. Complex networks of TclOO objects present a different problem: not only do we need to checkpoint the data contained within the objects, we need to be able to restore the actual network of objects. The requirement to destroy and create a network of small objects when loading a saved scenario is the reason why we have not represented simulation entities as instances of Snit types: the advantages are outweighed by the nuisance factor. But the requirement for polymorphism shifts the equation.

As a result, I defined the notion of a "Tcl bean". A bean is an object that can be easily checkpointed as a string, and easily restored. More than that, the entire collection of beans existing at any given time can be checkpointed and restored as a group. Along the way we were able to support cascading deletion of beans (i.e., deleting a bean automatically deletes all of the beans that it owns as well), cut and copy of beans, and undo of a variety of operations on beans.

The remainder of this paper will describe the bean architecture and implementation.

2. Behavior of an Individual Bean

A bean is essentially a fancy object-oriented record structure. Consider the following:

```
beanclass create address {
    superclass bean

    variable first
    variable last
    variable street
    variable city
    variable state
    variable zip

    constructor {} {
        next
        initialize instance variables...
    }

    methods...
}
```

Beans represent scenario data, which is to say user-editable data. Thus, given an instance of `address` the application can set and get its variables:

```
% set a [address new]
::bean::address1
% $a set first John
John
% $a set last Doe
Doe
% $a get first
John
```

The `set` and `get` methods operate on validly defined scalar instance variables, i.e., any scalar variable that exists in the instance's namespace, and will throw an error if given any other name.

Because beans need to be saved and restored, we need to be able to retrieve a bean's state and restore it again. We assume that a bean's savable state is simply the contents of its scalar instance variables; we can retrieve this state as a dictionary by means of the `getdict` method and restore it using the `setdict` method.

```
% set dict [$a getdict]
id 1 first John last Doe ...
% $a setdict $dict
```

Note that in addition to the variables defined in the class definition, the bean also has an `id`, which is provided by the `bean` superclass. Beans are created and referenced by the user, and so we need an ID by which they can be referred to in the GUI and in user scripts. Thus, every bean has a unique ID, assigned on creation. This ID can be queried, but cannot be reset.

The bean's `set` method is the usual way to modify the value of a bean's instance variable, whether from outside or in the bean's own method bodies: it prevents the value of the `id` variable from being changed, and it can notify the application that some bean has changed (and hence, that there are unsaved changes). The `setdict` method uses the `set` method internally, and subclasses can override it to provide additional behavior.

3. Saving and Restoring Beans

The basic assumption made by the `bean` class is that the application has at most one scenario open at a time, that all beans are part of the scenario data, and that therefore all beans need to be saved and restored *en masse*. We use the `bean` class object itself as the manager for the set of existing beans. Thus, we can make a "checkpoint" of all beans as follows:

```
set checkpoint [bean checkpoint]
```

The checkpoint is a nested dictionary with the following structure:

```
idCounter -> counter
beans -> beanID -> serializedBean
```

A serialized bean is a list

```
class object beandict
```

Thus, in the example above, if the single address record for John Doe is the only existing bean, the checkpoint would look like this:

```
idCounter 1
beans     {
    1 {address ::bean::address1 { id 1 first John ...}}
}
```

The checkpoint string can then be saved to disk in any desired way, and restored as follows:

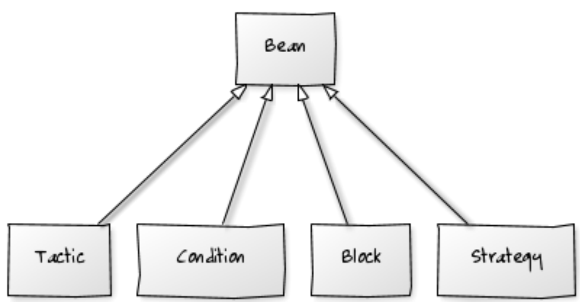
```
bean restore $checkpoint
```

The `bean restore` command ruthlessly destroys any existing beans, and uses the data in the checkpoint to recreate the saved beans. Note that the recreated beans not only have the same state, they have the same object names as before. The importance of this is discussed below in section 5.

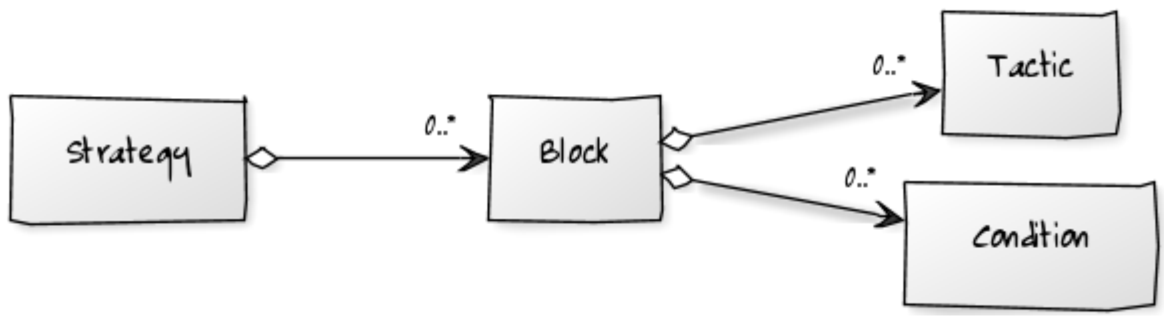
In order to act on all existing beans, the `bean` class object needs to know what beans exist. Consequently, the class object keeps a mapping from bean IDs to object names. This mapping is maintained by the `bean register` and `bean unregister` commands, which are called from the bean class's constructor and destructor respectively. It is also the `bean register` command's job to assign bean IDs.

4. Bean Trees

Beans can own other beans in complex tree structures. In Athena, there is a kind of simulation entity called an *actor*. An actor is a decision maker in the simulation. Each actor has a *strategy* which describes the actor's behavior. The strategy consists of a prioritized list of *blocks*; each block contains some number of Boolean *conditions* and some number of *tactics*. When the block's conditions are all met, the block's tactics will be executed. The strategy, blocks, conditions, and tactics are all implemented as bean subclasses; and there are many different kinds of condition and tactic.



We say that the strategy *owns* some number of blocks, and each block owns some number of conditions and tactics:



The relationships between the objects in a strategy are part of the scenario data, and need to be saved with the beans. Moreover, if an object is destroyed, the dependent objects (i.e., the tactics and conditions in a block) need to be destroyed with it. We would like these things to happen with minimal effort on the part of the application programmer; therefore we define the notion of a *bean slot*, an instance variable whose value is a list of zero or more names of bean objects that are owned by the object.

Thus, the block class is defined (in part) as follows:

```
beanclass create block {
    superclass bean

    beanslot conditions
    beanslot tactics
    ...
}
```

The `beanslot` command is simply a proc in the `oo::define` namespace. It verifies that the class for which it is called is truly a bean class, declares an instance variable with the given name to contain bean references, and then registers the slot name with the `bean` class object. In this way the `bean` class object's methods have enough information to walk the ownership tree.

Note that not all references to other beans need go in bean slots. For example, each instance of the `block` class keeps a reference to its owning `strategy`. Since the block does not own the strategy to which it belongs, the reference can be stored in an ordinary instance variable.

Thus, when destroying a bean it is straightforward to destroy the beans it owns, and the beans they own, and so on. There is no need for a bean subclass to write any specific destructor code to enable this behavior; the `bean` class's destructor handles it automatically.

5. Bean Object Names

Bean ownership and bean slots are the reason that it is necessary that bean object names be saved and restored: it allows bean objects to reference other bean objects by object name. Unique block IDs could be used instead, but this would require doing ID lookups before calling methods on referenced beans. Using object names keeps the code more concise and more readable. However, we must ensure that saving and restoring the set of beans cannot result in any object name collisions.

Beans can be assigned a unique name automatically by the bean class’s `new` method. By default, TclOO objects created using the `new` method are given names like `::oo::Obj12`. Beans can be created at one time, saved, and restored at another into a program with a different execution history. As a result, automatically named beans have to have names that won't conflict with non-bean TclOO object names.

This is why bean classes are created using the `beanclass` metaclass. It redefines the new method to create names that look like “::bean::*<class><id>*”, e.g., `::bean::address1`. Since only bean objects have names in the `::bean` namespace, and since all previously existing beans are deleted on `bean restore`, there is no chance of a name collision involving automatically named beans.

The application can also choose to create beans with specific names using the bean class’s `create` method. This is sometimes appropriate for top-level beans like strategies. Strategies are owned by actors (which are *not* beans); each actor has a short, well-known name and a single strategy, so strategy beans can be given names like “::strategy::*<actor>*”, e.g., `::strategy::JOE`. It is up to the application to make sure that explicitly named beans will cause no name collisions on `bean restore`.

6. Deleting and Undeleting Beans

Because beans are scenario objects, users can delete them as they edit the scenario. It is common for beans to own or contain other beans, and these dependent beans must be deleted with their owner; this called a *cascading delete*. And since the user can undo deletions, the Tcl Bean infrastructure has to support that as well.

A bean is deleted by passing its bean ID to the `bean delete` command; bean IDs are used because beans are usually deleted due to user input, and user input usually results in an ID rather than an object name. The command returns a *delete set*: a checkpoint-like string that allows the deletion of the bean and its dependents to be undone (under the usual conditions for a successful undo operation):

```
set delset [bean delete [$bean id]]
...
bean undelete $delset
```

The `bean delete` command starts with the given bean and walks the ownership tree destroying beans. The delete set is accumulated with the help of the `bean unregister` command, which is called during bean destruction, and is returned to the caller. The delete set is simply a dictionary

```
    beanID -> serializedBean
```

7. Cut, Copy, and Paste

Copy and paste of beans is similar to deletion. When copying a block in a strategy, the user will want to copy the conditions and tactics it contains as well. It differs in that any pasted beans will be new distinct beans.

The ability to copy and paste is still a work in progress; however, copying is logically straightforward. Given a bean ID, the bean copy command walks the ownership tree, copying the class, ID, and state of each bean in the tree. Bean slot variables contain a list of bean object names; in the saved state, each such variable is modified to contain a list of bean IDs instead. The result is called a *copy set*; it can be used to create a new set of beans at any time:

```
set copyset [bean copy [$bean id]]
...
set newbean [bean paste $copyset]
```

The variable newbean now contains the name of a new bean that is the root of the copied tree.

8. Other Operations

The bean infrastructure provides a number of additional operations. The `bean` class object allows the program to get a list of bean IDs, validate bean IDs, retrieve a bean given its ID, and so forth. Specific bean classes provide the same capabilities (via the `beanclass` metaclass) but limit them to beans of the appropriate type.

In addition to the `set` and `setdict` methods, each bean also has `lappend` and `ldelete` methods; thus you can write

```
    $bean lappend items $myitem
```

instead of

```
set list [$bean get items]
lappend list $myitem
$bean set items $list
```

Finally, the bean class provides a set of commonly used *order mutator* methods. In Athena parlance, an order mutator is a command that updates a simulation entity given a validated set of parameter values and returns a command that will undo the change. (An *order* is a command that takes a set of parameters entered by the user, validates them, and calls a mutator to do the dirty work.) By convention, mutator methods have names ending with an underscore.

The `addbean_` method adds a bean to a bean slot. The following code adds a new block to a strategy's `blocks` slot, saving the undo script.

```
set undo [$strategy addbean_ blocks [block new]]
```

The `deletebean_` method is similar: it deletes a bean with a given ID from a bean slot. The following code deletes a block from a strategy, returning an undo script.

```
set undo [$strategy deletebean_ blocks [$block id]]
```

The `update_` method handles most other changes to a single bean. It takes two arguments: a list of bean variable names, and a dictionary of parameter names and values. It updates the value of each of the listed bean variables, only if the provided dictionary has a matching entry whose value is not the empty string. (An empty string indicates that the user didn't edit that variable.) Again, it returns an undo script.

9. Bean Constraints

It's been said that architecture is the set of constraints you choose to live with because they make your problem simpler. These are the specific constraints and assumptions made by the Tcl Beans infrastructure.

- Every bean has a unique ID
- A bean's savable state is contained in its scalar instance variables.
- Beans are saved and restored *en masse*.
- Bean object names are restored along with the bean data.

- Beans can own other beans; the ownership graph is a forest of trees. Owned beans are stored in bean slot variables.
- A bean's constructor can be called with no arguments to create a bean in the default state. (This is required by the restore/undelete logic.) A bean's constructor may have any number of optional arguments.
- Every bean class must be created using the `beanclass` metaclass, and must be a subclass (direct or indirect) of `bean`.

Naturally, some of these constraints might be relaxed over time. For example, it would be straightforward to allow array-valued bean variables to be part of the bean's savable state, or to provide for multiple simultaneous sets of beans that can be saved and restored individually. But these were the constraints we thought we could easily live within given our current needs, and so there was no need to complicate things.

10. Bean Implementation

The Tcl Bean implementation consists of three pieces of code: the `beanclass` metaclass, the `bean` class itself, and the `beanslot` `oo::define` command.

All bean classes are created using the `beanclass` metaclass, which provides custom `new`, `get`, `exists`, and `ids` methods for all bean classes. The most important of these is the new method, which ensures that bean names are created in the `::bean` namespace, as described in Section 5. The other three methods simply allow the application to query beans belonging to the class or its subclasses. For example

```
set block [block new]    ;# Returns ::bean::block<id>
set ids [block ids]      ;# Returns all block IDs
block exists [$block id] ;# Returns 1
block get [$block id]    ;# Returns $block
```

If it weren't for the naming requirement, `beanclass` probably wouldn't exist.

Every bean class must be created using the `beanclass` metaclass, and must be a subclass (direct or indirect) of `bean`. It would be helpful if `beanclass` could ensure the latter requirement automatically, but I've not figured out how to do that in a graceful way.

11. TclOO Is Not Snit

Tcl Beans is my first attempt to implement production code using TclOO, and as a long-time Snit [2] user I ran into some surprises and annoyances. While some of them might in theory be fixed in future versions of TclOO, others are simply consequences of TclOO's advanced capabilities and have to be lived with and worked around. I record them here as a starting point for some kind of Snit compatibility layer or other utility library.

I'm aware that Tcllib already includes the beginnings of a TclOO convenience library, `oo::util` [3]; however, I'm not using it. I want to understand TclOO reasonably deeply at the scripting level before I start depending on external convenience libraries.

11.1 Variable Declarations

In Snit, you can declare an instance variable and assign it an initial value in the body of the type definition. In TclOO you can't; instance variables can be declared in the class definition, but can only be initialized in the constructor or in some other method body. This is particularly annoying when adding variables to an object using `oo::objdefine`: there is no way to initialize them from within the object's private (i.e., unexported) code. You have to use a publicly accessible (i.e., exported) method.

11.2 Linkage Between Types and Instances

In Snit, an instance and its type are closely linked. In particular, type variables are visible without declaration in instance code. In TclOO, instances are only weakly linked to the class used to create them; an instance's class can be changed at any time. Hence, variables belonging to the class object are not usually visible in instance methods.

This may an unavoidable consequence of TclOO's power and flexibility. However, it would be useful to be able to declare class variable once in the `oo::define` script and access it without declaration within instance methods, even at the cost of not being able to subsequently give instances a different class.

11.3 Code Layout

TclOO leads to harder-to-read code modules than Snit, in my view. I care very much about code readability, and designed Snit to maximize it—given my particular (or perhaps peculiar) sense of code aesthetics. It's not surprising that another person's system doesn't give me quite what I want, and the following observations might be a sore point only for me.

In my code, a Snit type with instances usually looks something like this template:

```
snit::type mytype {
    # Lookup tables
    # Other type variables
    # Type constructor (executes initialization code on load)
    # Type methods
    # Instance variables
    # Option definitions
    # Constructor/destructor
    # Instance methods
}
```

It's a logical sequence, and the entire type is clearly one thing. When you get to the closing "}" you know you've seen all of it.

TclOO supports a class definition syntax with a Snit-like definition body; and this works well unless you want to define methods on the class object itself. Then you need to use something like this to get the

```
oo::class create myclass

oo::objdefine myclass {
    # class variables
    # class methods
    method init {} { # Initializes class variables }
}
myclass init

oo::define myclass {
    # instance variables
    # constructor
    # instance methods
}
```

It should be possible to add commands to the `oo::define` namespace that would allow the class variables and methods to be included in one body with the instance definitions; I believe the `classmethod` command in `oo::util` works this way. I'm not clear on just what the side-effects are, though, or what constraints it places on the finished class.

11.4 Metaclasses Are Not Inherited

This is a purely TclOO-related surprise, as there's no Snit equivalent.

Tcl Beans defines the `beanclass` metaclass so as to customize the `new` method to generate a name in a particular namespace, and then uses it define the `bean` class. It would seem that subclasses of `bean` would inherit this behavior automatically, but they don't. Every subclass of `bean` has to be explicitly created using the `beanclass` metaclass. This feels wrong to me, though I freely admit that I don't understand it well enough to have an opinion.

12. Future Work

Tcl Beans have been implemented as a part of an experimental redesign of Athena 5's strategy execution engine. This redesign is not yet complete. Thus, the following work remains:

- Complete the re-implementation of the strategy engine. This will likely result in additional features and tweaks to the bean infrastructure.
- Complete the infrastructure to support copy and paste, and use it in the GUI.
- Review the bean code in light of further experience with TclOO, and see if it can be streamlined or improved.

13. Conclusions

As currently implemented, Tcl Beans make it simple to use TclOO objects to contain user data in a document-centric application provided that only one document can be open at one time. Bean classes are trivial to implement, and all bean instances can be saved and restored as a set. The infrastructure supports many other useful patterns, including cascading deletes, undo of user changes, and copy and paste. These features are subject to a handle of constraints, some of which it might be possible to relax in future versions.

14. References

[1] Duquette, William H., "Type-Definition Objects", 12th Annual Tcl/Tk Conference, http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/37573/1/05-2409.pdf.

[2] Duquette, William, Snit Object Framework, found in Tcllib, http://tcllib.sourceforge.net/doc/snit.html.

[3] Kupries, Andreas, and Fellows, Donal, TclOO Utility Package `oo::util`, found in Tcllib, http://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/ooutil/ooutil.html

15. 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 Athena Stability & Recovery Operations Simulation (Athena) for the TRADOC G2 Intelligence Support Activity (TRISA) at Fort Leavenworth, Kansas.