Application to Library: Re-Architecting a Large Monolithic TCL Application

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

Abstract

The Athena Regional Stability Simulation was designed as a monolithic TCL/TK application supporting one simulation scenario at a time. The simulation and data management code—the core of the application—consisted of over fifty-thousand lines of code divided across many modules, many of which were singletons providing resources to the entire application. This code has since been extracted into a scenario class supporting any number of simulation instances within a single application. The resulting implementation uses a novel Snit-based approach to creating facades for complex object-oriented libraries.

1. Overview

The Athena Regional Stability Simulation is a large time-step simulation of stability operations in a war-torn region. The set of input data that describes the region and the actors within it is referred to as a *scenario*. For its first seven major releases, Athena was a monolithic TCL/TK desktop application supporting one scenario at a time. To address multiple scenarios simultaneously one had to invoke multiple instances of the application; and it was impossible to access Athena's models from within any other application.

In the fall of 2014 the Athena team received the requirement to support deployment of Athena as a cloud-based application with a web interface. JPL's role was to develop an Athena back end that could sit on a cloud node and interact with the wider web through a wrapper developed by another team.

In addition, the Athena team was tasked with improving Athena's presentation of its outputs, and in particular to make it much easier to compare the results of multiple simulation runs.

It quickly became clear that Athena's models needed to be accessible in multiple applications, and that any given application might need to load multiple scenarios simultaneously. After some discussion we determined that Athena's scenario management and simulation modeling code needed to be abstracted from the desktop GUI application into a library in the form of a class or Snit type [1] supporting multiple instances.

2. Initial Architecture

When work began, Athena had the architecture shown in Figure 1:

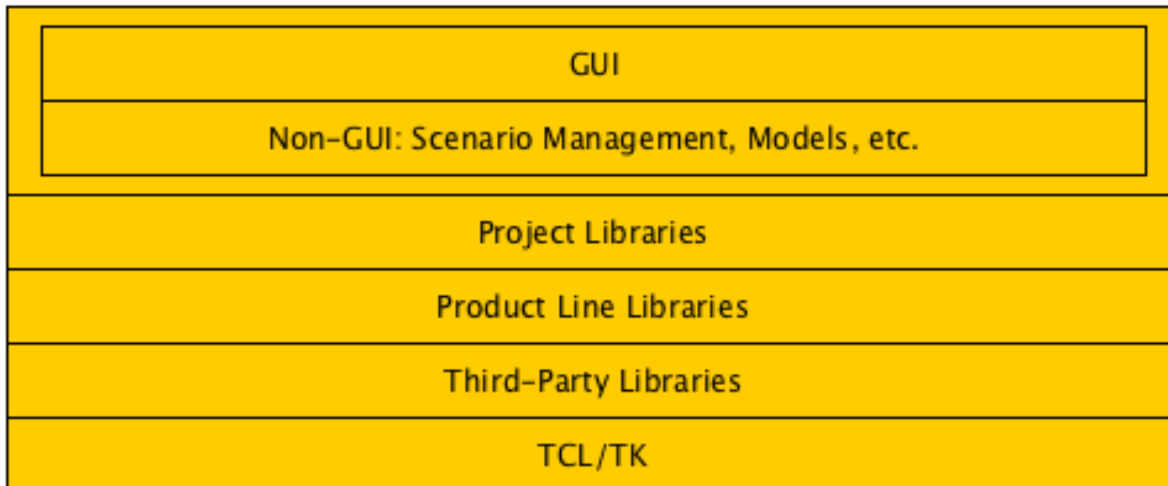

Figure 1: Layered Architecture

The application proper sat on a stack of libraries, many of which were written by the Athena team. The application itself was divided into two layers: the non-GUI layer, containing the scenario management and simulation management code, and the GUI layer on top of it.

It has always been our practice to push application code down the stack into libraries whenever possible. What has remained in the non-GUI layer, consequently, were those modules that were so interconnected that abstracting any piece as a separate library was likely to add complexity rather than reduce it.

The architecture of the non-GUI layer is shown in Figure 2. The non-GUI layer consisted primarily of a collection of singleton objects with well-known global names. In some cases a singleton was a Snit type singleton; in other cases it was an instance of a Snit type or TclOO class. The web of dependencies between these objects was (and is) complicated; giving them global names meant that every object could in principle call any of the other objects. In some cases a singleton might own and manage a collection of components in the usual object-oriented way; in such cases, the dependent objects were accessed via the owning singleton.

For example, the `::actor` singleton provided a palette of subcommands for managing the "actor" entities in the scenario. Any module in the application might need to call its subcommands to acquire information about one or more actors.

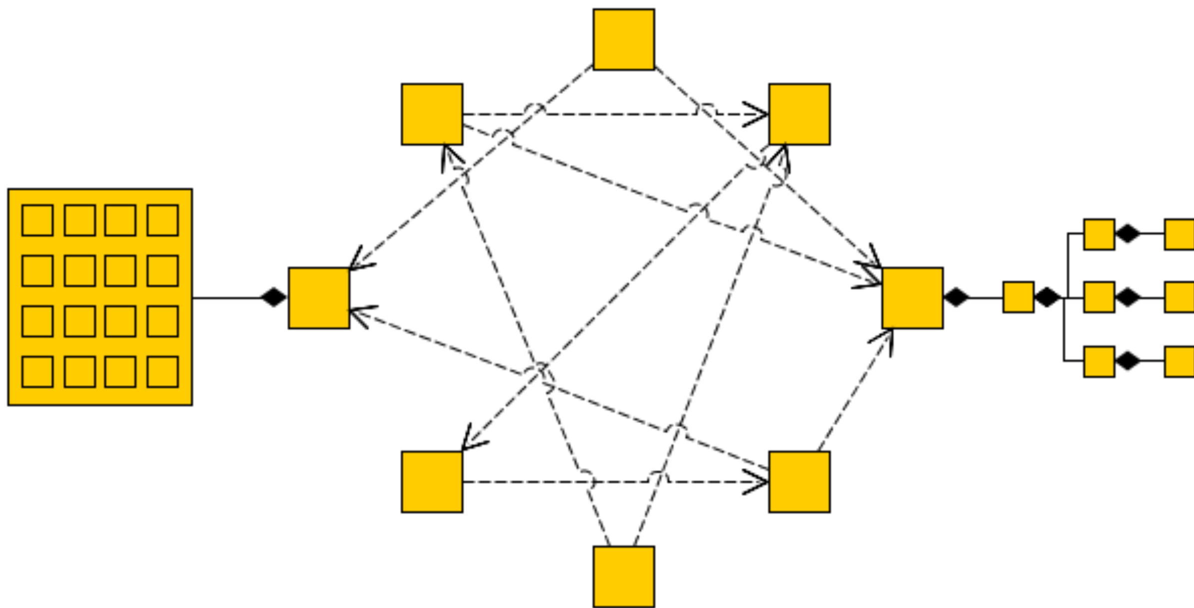

Figure 2: Singleton Objects

```
# Iterate over the actors
foreach a [actor names] {
    array set data [actor get $a]
    actor update $a . . .
}
```

Our task, then, was to abstract out the bulk of the non-GUI layer into a new library, structured as a class with multiple instances.

3. Architectural Requirements

We placed the following constraints on the new architecture; two of them have already been mentioned:

- The new library shall provide an Athena scenario class.
- The class shall support multiple simultaneous instances.
- Code written using the new scenario class shall be as concise and readable as the existing code, in so far as this is possible.
- The public API shall not directly expose private functionality.

4. Object-Oriented Design

A naive design for an object-oriented library is shown in Figure 3:

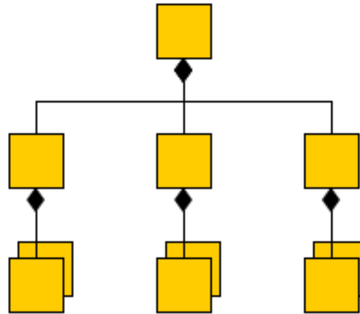

Figure 3: Naive Object-Oriented Design

The library provides a public class, shown at the top. That class defines a number of components; and each of those components may have components of its own. Coupling is minimized and cohesion is maximized if method calls flow along the ownership relationships.

We initially attempted to implement the Athena application in this style, and were quickly frustrated. A simulation like Athena is modeling the real world (albeit in a highly abstract way), and entities in the real world rarely fit into neat stovepipes. Similarly, the modeled entities link to each other in manifold and surprising ways (Figure 4):

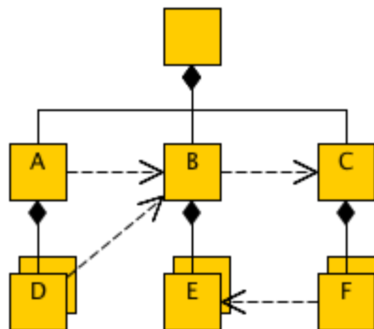

Figure 4: Naive Object-Oriented Design with Dependencies

For example, object A has to know how to find object B, which has to know how to find object C; object D has to access object B through A, and object F has to access object E through C and B. Worse, these links change and evolve as the models change and evolve, and early in the development process the models were evolving rapidly. We were under considerable time

pressure, and managing the plumbing along the above lines proved to be a full-time job. We quickly abandoned this naïve architecture for the simplest thing that could possibly work, which turned out to be the singleton-based architecture shown in Figure 2. That architecture served us well for six years.

Given time to think and greater experience with the problem, we realized that the difficulty with the naïve infrastructure shown in Figures 3 and 4 is that the major modules of the simulation are truly all peers and all need ready access to each other. Instead of manually plumbing each interconnection we needed a kind of bus architecture. The singleton-based architecture provided that kind of access: the entities on the bus were the singletons, and the messages were simple ensemble calls.

Such a bus architecture can be implemented in an object-oriented way as shown in Figure 5.

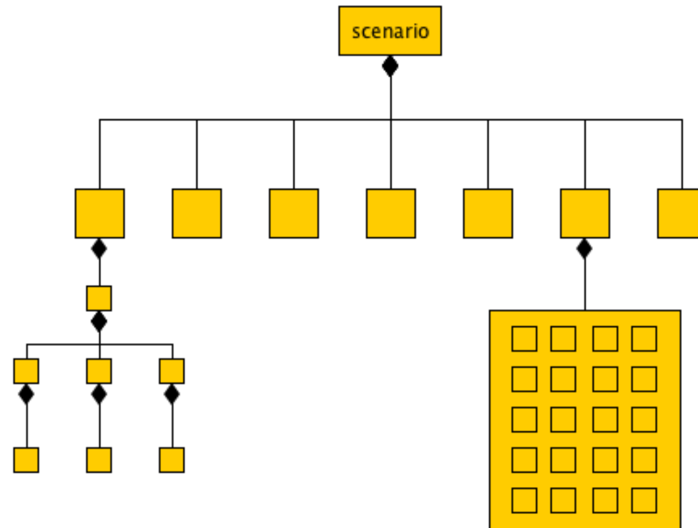

Figure 5: Peer Architecture

The scenario object is the master object, and the primary interface to the simulation and scenario management code. The major modules all become direct components of the scenario object: that is, each singleton becomes an instance of a type or class, and that instance is created and owned by the scenario object. Those modules that own managed subcomponents go on owning and managing those subcomponents, just as before. Then, interaction between the peers is handled as shown in Figure 6.

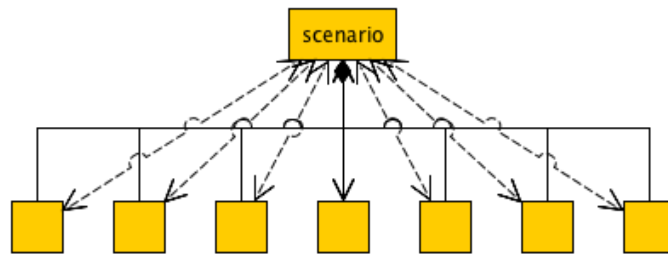

Figure 6: Peer Interaction

The scenario object itself serves as the bus. It passes itself to each component's constructor as it creates the component; thus, every component has access to the scenario, and the scenario has access to each component. The peers interact with each other via the scenario object and the scenario object provides a façade over the rest of the library.

5. Implementation of Peer Components

The next problem was implementing this bus architecture. The existing code was concise, readable, and maintainable; we wanted the new implementation to lend itself to code that was no less concise, readable, and maintainable (insofar as that is possible with a more complex architecture). Also, we sought a solution that allowed us to transition the code base incrementally, and with minimal changes to the existing code—especially as regards the automated test suite.

5.1 A Naïve Implementation

There are many ways to implement a bus architecture, including an explicit message passing infrastructure; within a single library such as this, the simplest mechanism is simply the direct method call. The complexity lies not in the message but in identifying and accessing the object on which the method will be called.

In a language like C++ or Java one might simply make the peer components public members of the scenario object, and access a component's methods using dot-notation. In Java, one might iterate over the names of the actors in the scenario like this:

```
for (String name : scenario.actor.names()) { . . . }
```

More likely, one would provide a scenario method to retrieve the actor component:

```
for (String name : scenario.actor().names()) { . . . }
```

A naïve implementation of the Java/C++ pattern in Tcl leads to code like this,

```
set actors [scenario actor]

foreach a [$actors names] { . . . }
```

or more concisely,

```
foreach a [[scenario actor] names] { . . . }
```

This is ugly on the face of it, and the problem only gets worse with each level of indirection. Unfortunately, Tcl lacks any concise, readable way of chaining object references in situations like this…or does it?

5.2 A Tclish Implementation: Snit's Public Components

In fact, this pattern arises quite frequently in the Tk widget set. The text widget manages a plethora of objects, including text snippets, marks, and tags. The widget provides access to these managed objects by means of a hierarchical command set. Tags, for example, are accessed using subcommands of the widget's **tag** subcommand. In calling **$w tag add** we have implicitly accessed the text widget's tag manager component.

It so happens that this pattern is supported quite readily by an obscure feature of Snit's delegation capability: the public component.

An instance of a Snit type can define any number of components using the **component** type definition statement, and can delegate its own subcommands to these components. Further, a component can be declared to be "public":

```
component actor –public actor
```

This statement says that the scenario object has a component called "**actor**", and that the actor component is to be made public as a subcommand also called "**actor**". Assuming that the variable **scn** contains the name of the scenario object, then, we can write code like this:

```
# Iterate over the actors
foreach a [$scn actor names] {
    array set data [$scn actor get $a]
    $scn actor update $a . . .
}
```

Note that this is almost exactly the same code we started with, except that we've inserted the **::scn** command at the beginning of each erstwhile **::actor** command.

This gives us our implementation. Each peer component becomes a public component of the scenario object. Since each peer component knows the name of its owning scenario object, each component has seemingly direct access to every other peer component.

This technique extends to any number of levels; if the **`actor`** component defines a public component, it can be accessed as a subcommand of the **`actor`** subcommand of the **`::scn`** object.

5.3 A Peer Component

Peer components can be implemented as instances of Snit types or TclOO classes, or as any kind of *ad hoc* ensemble that receives and uses the scenario object. Here is a typical skeleton:

```
snit::type ::athena::actor {
    component scn  ;# The scenario object

    constructor {scn_} {
        set scn $scn_
        . . .
    }
    . . .
    method names {} {
        # return the list of actor names
    }
    . . .
}
```

The peer's code can use the **$scn** component to access any other peer, as well as any services provided by the scenario object itself. Further, if the peer component has subcomponents it can pass **$scn** down to its subcomponents and thereby give them access to all of the peer components.

5.4 The Scenario Object

The scenario type is implemented as a Snit type. In addition to managing all of the peer components, it also contains the code for saving and loading scenario data and for controlling the execution of the simulation. Here is the basic skeleton that shows how the components are managed:

```
snit::type ::athena::athenadb {
    # Components
    component actor -public actor
    component group -public group
    . . .

    constructor {. . .} {
        . . .
        $self MakeComponent actor
        $self MakeComponent group
        . . .
    }

    method MakeComponent {comp} {
        install $comp using ::athena::${comp} ${selfns}::$comp $self
    }
    . . .
}
```

In particular,

- The components are defined as types or classes in the **`::athena`** library.
- The types or classes have the same name as the component.
- The component objects are created in the scenario object's private namespace, **`$selfns`**; as a result they will be destroyed automatically when the scenario object is destroyed, with no need for explicit destructor code.
- The component objects are passed the name of the scenario object itself.

There are special cases, but the majority of the components are created exactly as shown. And all such components can access each other as subcommands of the scenario object.

5.5 The Public Interface

The scenario object described in Section 5.4, **`::athena::athenadb`**, is the *private* scenario interface, the interface used by the various components of the scenario to interact with each other as the user edits the scenario and as the simulation runs. By the nature of things, this interface exposes many methods that clients of the **`::athena`** library cannot call directly without the danger of violating public invariants. Consequently, we also needed to define a public interface for use by clients of the library.

One possibility was to expose the scenario type described above, but to only document those components and subcommands that are suitable for public use. We rejected this, as it makes it too easy for the client to get into trouble. It's fine for Tcl libraries to give the client enough rope to hang himself, but the rope should be stored in a conveniently available closet rather than

dangling from a tree limb. Consequently, we layered the public scenario type, **`::athena::athena`**, as a wrapper around **`::athena::athenadb`**, the private scenario type, as shown in Figure 7:

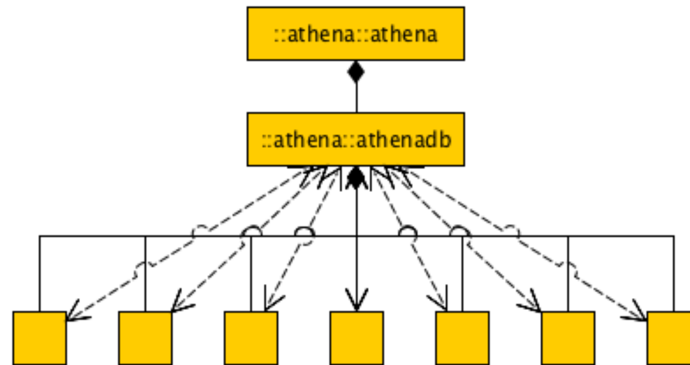

Figure 7: Public vs. Private Façade

The public façade is no more than a shell. It has one true component, an instance of the private scenario type, and delegates all of the public options and methods to that component. In this way we were able to build up an object providing only the desired public interface with minimal code. The skeleton looks like this:

```
snit::type ::athena::athena {
    . . .
    component scn    ;# The instance of ::athena::athenadb

    delegate options -subject to scn
    . . .

    constructor {. . .} {
        install scn using athenadb ${selfns}::scn . . .
    }

    delegate method {actor get}   to scn as {actor get}
    delegate method {actor names} to scn as {actor names}
    . . .
}
```

Instead of making the components public, we explicitly delegate the required subcommands one by one.

5.6 Managing the Transition

The Athena application's non-GUI layer consisted of over 50,000 lines of code spread over more than one-hundred modules; converting it all to the new architecture required weeks of effort. Fortunately, we were able to do the conversion incrementally.

First, we wrote a skeleton scenario type that instead of creating its own components simply declared the application's existing singleton objects as its public components, e.g.,

```
snit::type ::athena::athenadb {
    component actor -public actor
    component group -public group
    . . .

    constructor {. . .} {
        . . .
        set actor ::actor
        set group ::group
        . . .
    }
}
```

This allowed an instance of the type to be the interface to the existing application's code while leaving the old interface in place. We were still limited to working with a single scenario at a time, but it allowed us to begin to update the code base to use the new interface. This skeleton scenario type became the first module in the new **`::athena`** library.

Next, we revised the Athena application to create an instance of the new scenario type, and then revised the Athena application's GUI layer to refer to the scenario data only via this instance.

Next, we began moving modules from the Athena application's non-GUI layer to the new **`::athena`** library and converting them to be true component objects of the scenario type, updating references as we went.

In this way we moved from the old architecture to the new architecture over a period of months, while retaining the ability to run and test the application throughout the process. As the project includes over 60,000 lines of tcltest(n) test scripts, this ability was crucial to ensuring that we remained on the right track.

The above description covers a multitude of sins; for example, there were a number of library modules that expected to be global resources that also needed to be converted to a proper object-oriented architecture, and these conversions formed at least half of the total work. Once the new architecture was defined, however, the conversion proceeded smoothly along the lines described here.

6. Results

6.1 Interface

The public scenario type, which is defined as a Snit type called **`::athena::athena`**, has over sixty subcommands, many of which have six to ten subcommands of their own. The private scenario type, which is defined as a Snit type called **`::athena::athenadb`**, adds a few more subcommands and adds additional subcommands to many of them. These are certainly the most complex Snit types (in terms of the interface) that the author has yet seen.

We've been working with the new code for about six months, and are pleased with the results. The simulation modules are no harder to maintain or test than they were previously, and encapsulating an entire scenario as a single object has been tremendously freeing.

6.2 Performance

We were concerned that the additional layer of subcommands would cause a significant performance hit, but this turned out not to be the case. The new code is neither obviously slower nor obviously faster than the old code.

6.3 Snit vs. TclOO

Athena currently uses a mixture of Snit and TclOO code. We have found that TclOO is the better fit for small, numerous, lightweight objects, especially when inheritance is required. Nevertheless, Snit continues to hold its own in a number of areas. It remains the megawidget framework of choice, and it is much more convenient when defining complex interfaces such as those described in this paper due to its support for hierarchical command sets and its sophisticated delegation support. (TclOO provides the "forward" statement, but it's more of a do-it-yourself kit than a full solution.) Finally, the tight coupling between Snit types and their instances makes it easy to handle related collections of objects in an easy and efficient way.

TclOO can certainly be made to perform all of these functions; but Snit holds the edge in terms of brevity and readability.

7. References

[1] Duquette, William, Snit, found in Tcllib, http://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/snit/snit.html.

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 Athena Stability & Recovery Operations Simulation (Athena) for TRADOC G2-7 at Fort Leavenworth, Kansas.

Copyright 2015 California Institute of Technology. Government sponsorship acknowledged.