# Anatomy of a Large Application: Architectural Patterns and Solutions

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

## ABSTRACT

JNEM, the Joint Non-kinetic Effects Model, is a large simulation application. Written almost entirely in Tcl/Tk, it makes architectural use of the Snit object system and the SQLite3 database engine. This paper addresses a number of architectural patterns and solutions that have been found useful during the two-plus years of JNEM development. Patterns include the three-layer package architecture (application, domain, and utility); Snit types as application modules; saving and restoring application state; the database-backed objects; SQLite3 as an application memory debugger; and a generalization of the scrollbar/scrollable pattern.

## 1. Joint Non-kinetic Effects Model

The Joint Non-kinetic Effects Model (JNEM) is a military training simulation that participates in a federation of simulations used to train military commanders. The federation is called the Joint Land Component Constructive Training Capability (JLCCTC) Multi-Resolution Federation (MRF). JNEM's role as a federate in the federation is to model the responses of the civilian population to force activities, up to and including actual combat, thus adding non-kinetic effects to the kinetic effects modeled by the battlefield simulation. JNEM is written primarily in Tcl/Tk 8.4, with a small amount of code in C/C++.

This paper describes certain architectural patterns and solutions used in the implementation of JNEM; it does not address specifics of the JNEM implementation, simulation model, or data formats. The techniques described here can be adapted to any large application/system with similar requirements.

## 2. The Three-Layer Package Architecture

Like many large application development efforts, JNEM is not really a single application; rather it is a system containing many applications of various sizes, of which the JNEM simulation proper is the largest. The system also includes control GUIs (the JNEM Console), simulators designed to represent other federates during development, and a number of ancillary tools of varying degrees of complexity, both GUI and non-GUI. As a result, there is opportunity for significant code sharing between the various components of the system.

The cleanest mechanism for sharing code between applications is the use of well-written reusable code libraries. Reusability, however, is context-dependent. Every coding effort is based on assumptions as to the context in which the resulting software will be used. This context then determines the arena in which the software is reusable.

The wider the context in which the code is to be reused, the more general and flexible (and hence the more complicated and costly) the code needs to be. The narrower the context, the more assumptions the author of the code may make and the more specific (and hence simpler and less costly) the code may be.

It is important to target code to the proper context. If a library module is written for too broad a context, it will be more complex than necessary, and thus more costly to implement and maintain. The increased complexity is often reflected in the module's interface; this in turn increases the complexity of all clients of the module, thus increasing the cost of using the module.

If the library module is written for too narrow a context, then it may not be possible to take advantage of unforeseen opportunities for reuse that will arise. When this happens, either the library module needs to be generalized—if there is time—or the coder is likely to be forced to engage in "bad reuse": copying the library code into a new module with small changes.

Deciding the appropriate context for a new module of code is to a certain extent an art, and depends on a clear vision of what the project's needs will be in the future. However, we have found that the following three-layer package architecture greatly simplifies the decision:

| Application Layer |
| --- |
| Domain Layer |
| Utility Layer |

Each layer represents a particular context and has its own particular constraints; generality increases from top to bottom. The layers are defined in detail below; once they have been defined, we will discuss how code is structured within each layer.

### 2.1 The Utility Layer

The utility layer represents the widest context in the system. Code written for this layer should make few assumptions about the environment in which it runs, and, in particular, should avoid placing requirements on the application. For example, JNEM's utility layer includes a pair of comm(n)-based communications modules, commclient(n) and commserver(n). Among their duties is the logging of client connects, disconnects, and other message traffic. However, they write to a log file only when configured to do so; otherwise they would be suitable for large, long-running applications but not for short *ad hoc* utility scripts.

These considerations place the following constraints on modules in the utility layer:

**Modules should not touch the global namespace.** All utility layer code should be defined in package namespaces, with public names exported.

**Modules without state should usually export a single ensemble command.** This was determined to be a beneficial pattern by Roseman [3], and minimizes the risk of name collisions if names are imported into the global namespace. There are exceptions; frequently-used utility commands can be implemented as normal commands. But generally speaking, a family of related commands should be implemented as an ensemble. For example, JNEM's family of matrix operations are implemented as a single ensemble, mat(n).

**Modules with state should be implemented as object instances**. JNEM's logging facility, for example, is defined as a logger(n) object type, thus allowing an application to create multiple logger(n) objects and log to multiple files simultaneously.

**Object instances must be wired together explicitly.** If a commserver(n) object is to log communications traffic, for example, it must be given the name of a logger(n) object. It cannot simply assume that there is a logger(n) with a well-known name.

JNEM's utility layer includes the following kinds of modules:

- String and list handling
- Date and time
- Interprocess communication
- Math and geometry
- Control structures
- Data types
- Logging
- General purpose megawidgets

A module that contains no domain-specific knowledge should be included in this layer on one of two conditions: if it is definitely needed by two or more applications within the system, or if it has potential for reuse and little extra work is required to make it a library module. The logger(n) module is an example of the first case; string and list handling routines are examples of the second.

JNEM is not a cross-platform application; it specifically targets Linux/X-Windows. If it were to be made cross-platform, platform details would be handled in this layer as well.

## 2.2 The Domain Layer

The domain layer contains code intended for reuse in multiple applications in our particular application domain. For example, JNEM defines a module called sqldatabase(n), a Snit-based wrapper for SQLite3 database objects. Among other features, the wrapper defines the JNEM database schemas. This information is clearly specific to our application domain, and hence does not belong in the utility layer; on the other hand, we wish to be able to write a variety of applications, from the JNEM simulation proper down to simple scripts, that access JNEM database files. Consequently this code belongs in the domain layer, along with domain-specific data types, file format definitions and parsers, multi-application simulation modules, and domain-specific multi-application GUI components. As with the utility layer, code written for this layer should avoid placing requirements on the application.

Domain layer code generally operates under the same constraints as utility layer code; however, it will include domain-specific knowledge and will be more likely to implement specific policies. JNEM's utility layer, for example, includes a module called sqlib(n), which contains routines for querying SQLite3 schemas and for formatting the results of SQLite3 queries. This module makes no assumptions about the SQLite3 database upon which it operates, and defines no policies for how the database is used. The domain-layer module sqldatabase(n), on the other hand, not only defines the database schema but also implements a document-like open/save transaction policy.

A module containing domain-specific knowledge should be defined in this layer on either of two conditions: if it is definitely needed by two or more applications within the system, or if it has potential for reuse and little extra work is required to make it a library module.

It sometimes happens that a module requiring domain-specific infrastructure would also be useful in programs based solely on the utility layer. In this case, one needs to consider generalizing the module and its interface so that the module itself contains no dependencies on the domain layer, but can be configured to make use of domain-specific modules when available.

## 2.3 The Application Layer

The application layer contains code that resides in a single application. Properly speaking, then, there is no system-wide application layer; each application within the system has its own. Code written for this layer may access any desired modules from the two lower layers, and may embody any assumptions that simplify the implementation. The application may use the global namespace freely, and many objects in the application layer will have well-known names. For example, the application may define a global logger(n) object called `::log`, and all application modules may use it freely by that name. No "wiring together" of objects is required.

We have found the following guidelines to be helpful.

**Define public names in the global namespace.** We are used to not polluting the global namespace in our code, so as to avoid name collisions. But if the application itself cannot use the global namespace, who can?

**Define private names in namespaces.** A module's private internal commands should be defined in a module-specific namespace, so as to avoid inadvertant inter-module name collisions.

**Prefer singletons to object types with multiple instances.** Application modules will frequently contain state. Unless multiple instances of that state are definitely required, however, modules should be implemented as singleton objects with well-known names. This simplifies the code, and makes it easier to call modules from other modules without explicit wiring.

**Implement singletons as ensemble commands.** This gives each singleton module a single well-known name, and allows it to be treated as an object by other modules. The benefits of using ensemble commands for module public interfaces are discussed in [3].

## 2.4 Internal Architecture

Since the utility and domain layers consist of library modules, it will come as no surprise that each is implemented as a collection of packages. In principle, each module in these layers could be implemented as a separate package. For simplicity, however, JNEM groups the modules into a small number of packages, as shown below, although each module has its own man page:

| | GUI | Non-GUI |
|---|---|---|
| Application Layer | jnem_*app*(n) | |
| Domain Layer | simgui(n) | simlib(n) |
| Utility Layer | gui(n) | util(n) |

In use, the primary distinction is between GUI and non-GUI code, as requiring the Tk package in a non-GUI application has undesirable effects. Consequently, the utility and domain layers are each defined as a pair of library packages, one GUI and one non-GUI, each of which contains a number of modules.

Two architectures are used at the application layer: small applications, such as command-line utilities, are generally implemented as simple scripts which load the packages they need from the domain and utility layers. Larger applications, such as the JNEM simulation proper, are implemented as a short loader script which loads an application package, jnem_*app*(n). This architecture allows the application to contain arbitrarily many modules in an easily managed way. In JNEM, all such applications share a single loader script, jnem(1). Thus, the command

```
$ jnem sim
```

causes the jnem(1) script to load the jnem_sim(n) package and invoke its main entry point. By convention, every jnem_*app*(n) package contains at least one module, `app.tcl`, which defines a singleton object called `::app`. The application's main entry point is then `::app init`.

## 3. Snit Types as Application Modules

As stated above, an application module should present its public interface as an ensemble command defined in the global namespace, while the module's internal code should reside in a module-specific namespace. A `snit::type` definition serves this purpose admirably well:

- The type's name, defined in the global namespace, is the ensemble command.
- The type's type methods are the ensemble command's subcommands.
- All of the type's code and variables naturally reside in the type's namespace.

In addition, ensemble subcommands can be delegated to component objects or to other application modules, and ensemble subcommands can themselves be ensembles with subcommands of their own.

The one necessity is to ensure that the `snit::type` cannot create instances; otherwise, a mistyped subcommand will be treated as the name of an instance to be created, with mystifying results. The skeleton for such a module looks like this:

```
snit::type mymodule {
    pragma -hasinstances   no

    typevariable myvariable

    typemethod mysubcommand {args} {
        ...
    }
    ...
}
```

The `::mymodule` command is global, but all of the module's code then resides in the `::mymodule::` namespace.

The type's standard "info" and "destroy" methods can also be disabled, allowing them to be redefined by the module if the author so desires.

## 3.1 Library Ensembles

Ensemble commands implemented in the utility or domain layers can also be implemented in this way, placing the command in the package namespace. JNEM's matrix manipulation module, mat(n), a module of package util(n), is implemented something like this:

```
namespace export ::util::mat

snit::type ::util::mat {
    pragma -hasinstances   no

    typeconstructor { namespace import ::util::* }

    typemethod add {mat1 mat2} { ... }
    ...
}
```

The ensemble command is defined in the `::util` namespace, as are util(n) commands. Note that the ensemble's code resides in the `::util::mat` namespace, and consequently cannot see other commands defined in `::util`; hence, the ensemble must import them as shown. This is a nuisance, as it constrains the order in which util(n)'s modules are loaded. If JNEM were using Tcl 8.5 and Snit 2.1, this would not be necessary; in Snit 2.1, Snit types automatically add their parent namespace to their namespace path.

## 4. Saving and Restoring Simulation State

Training exercises can run twenty-four hours a day for five days or more. If a simulation in the training federation should crash, it is vital not only to get it running again as soon as possible, but also to get it running again with the same state it had prior to the crash (possibly adjusted slightly so as to avoid crashing again). Now, the state of a simulation is a complex thing, and the only way to recreate it from scratch is to re-run it with the same inputs and random draws. After more than a few hours, though, the time involved in "running up" the simulation from scratch is prohibitive. Consequently, every simulation in the federation is periodically asked to save its state so that the federation state can be restored later. Such a saved state is called a *checkpoint*, and saving simulation state is referred to as *checkpointing the simulation*.

In one sense, this is no different than the requirement on any document-centric application—a word processor, say, or a

spreadsheet. There are, however, two distinctions of note. First, the state data for a complex simulation can be of great variety and extent. In a word processor, each document is likely to be represented as an object of some type, probably managing a hierarchy of lower-level objects; and saving the document to disk is a natural operation on the document object. In a complex simulation, there are likely to be many, many objects and ancillary data to be saved, and there might not be any obvious organizing principle corresponding to the notion of a "document". Second, the requirement to load and save state is implicit in the notion of a document-centric application; it is not similarly implicit in the notion of a simulation. History shows that checkpointing is an architectural issue, and that it can be very difficult to implement if it isn't taken into account from the beginning.

Code written to save and restore complex application state to and from disk can be extremely fragile. First, all relevant state must be identified; if any state variables are omitted, it will not be possible to restore the simulation's state precisely. We will refer to this set of checkpointed state variables as the application's *persistent state*. Second, the routines which read the checkpoint must exactly mirror the routines that write it out, or efforts to restore will fail. An error in either one renders the checkpoint useless; and naturally, it's a common error (especially in applications for which checkpointing is an afterthought) to update the reader and forget to update the writer, or *vice versa*. The solution is to provide a framework for saving and restoring arbitrary state variables, and then register all state variables with the framework.

In most languages, it would also be necessary to register the type of each state variable. In Tcl, where everything is a string, defining such a framework is nearly trivial. JNEM follows a few simple patterns which make saving and restoring simulation state both easy to implement and robust in the face of change.

JNEM's internal state is of four kinds:

- Persistent state stored in the *run-time database* (RDB)
- Persistent state stored in Tcl variables which are mirrored in the RDB.
- Persistent state stored in Tcl variables which are not mirrored in the RDB.
- Non-persistent state stored in Tcl variables.

Only the first three kinds need to be included in a simulation checkpoint; however, the latter three kinds need to be clearly commented and distinguished in the code.

## 4.1 Saving the Run-time Database

The run-time database, or RDB, is an SQLite3 database. Much of JNEM's simulation data is stored there, particularly its knowledge of the various entities in the simulated world. Checkpointing the data stored in the RDB is, of course, trivial—JNEM creates a checkpoint simply by committing all current updates and making a copy of the database file. The data is restored by copying and opening the checkpoint file as a new RDB.

The checkpoint file is thus an SQLite3 database file. If all persistent state were stored in the RDB, no further work would need to be done. Inevitably, for performance or ease of implementation, some data will be stored only in memory. Some state data, notably object instance data, is stored in memory and automatically mirrored on change in the RDB; this makes it easy to perform queries on objects and also to produce reports. Such data is automatically checkpointed. Other state data in Tcl variables gets copied to and from the database on checkpoint and restore, as discussed in the following section.

## 4.2 Saving In-Memory Application State

The simulation consists of a number of modules, each of which has its own set of Tcl variables, some of which contain persistent data and some of which do not. Each module that has Tcl variables containing non-mirrored persistent state is registered with the checkpoint management module. This registry is simply a hard-coded list of module names; it is updated by hand as modules are added and deleted.

Each such checkpointable module is required to have the two following subcommands:

*module* `checkpoint`
: Returns the module's non-mirrored persistent state as a single Tcl value. This is usually a dictionary, and frequently a dictionary of dictionaries.

*module* `recover` *state*
: Restores the module's state to *state*, which must be a value returned by the module's `checkpoint` subcommand.

When a checkpoint is to be saved, the checkpoint manager simply asks each checkpointable module for its state, and stores it under the module's name in an RDB table called `checkpoint`. which has two string-valued columns, `component` and `data`. When saving a checkpoint, the application retrieves the state for each checkpointable module and stashes the module's name and state data into the `checkpoint` table. When restoring from a checkpoint, the application retrieves the module names and state values from the restored RDB file and asks each module to recover itself.

The preceding discussion uses the term "module", but some of the checkpointed "modules" are actually instances of `snit::type`s. What they all have in common is that they all have well-known global names, are created at simulation start, and persist for the life of the simulation. (Transient objects, on the other hand, mirror their state to the RDB.) Thus, the code is extremely simple; the only maintenance that is ever needed at the application level is to add and delete names from the list of checkpointable "modules" as the simulation's implementation changes during the course of development.

## 4.3 Saving In-Memory Module State

The previous section explained how the application saves and restores the persistent state of its modules to and from the RDB using each module's `checkpoint` and `recover` subcommands. This section explains how each module structures its internal data, and how the `checkpoint` and `recover` subcommands are usually implemented.

First, each checkpointable object is either a singleton implemented as a `snit::type`, as described above, or an

instance of a `snit::type`. Either way, the object's variables are grouped and clearly labeled as to whether they are checkpointed or not. Second, almost all checkpointed data is stored in one or more Tcl arrays. Some data is array-oriented by nature; and the remaining scalar data is stored in an array simply to make checkpointing convenient. The `checkpoint` and `recover` subcommands can then be implemented like this:

```
typemethod checkpoint {} {
    list \
        array1 [array get array1] \
        array2 [array get array2]
}

typemethod restore {checkpoint} {
    foreach {name value} $checkpoint {
        array unset $name
        array set $name $value
    }
}
```

In a few cases an object might have component objects with persistent state; this complicates the code slightly. But in each case, the pattern is the same: the object's state is checkpointed as a dictionary of named values and recovered accordingly.

The beautiful thing about this mechanism is that it is only necessary to touch the checkpoint/recover code when a checkpointed array or component is added to or removed from an object. Since checkpointable scalar variables are grouped into an array, any number of new scalar variables can be trivially added and correctly checkpointed just by defining them in that (carefully labeled) array.

In short, simply by storing data in arrays and adopting a simple convention, we get trouble-free saves and restores of in-memory data with almost no maintenance overhead. It just works.

## 5. Database-Backed Objects

The word "object" being oversubscribed, we will adopt the following terminology for this section. An *object* is a standard Tcl object: a command with subcommands. A *singleton* is an object defined as a `snit::type` with type methods. An *instance* is an object defined as an instance of a `snit::type`. An *entity* is a simulated thing with associated data that might or might not have an associated *object*.

In JNEM, for example, a ground unit—a platoon, say—is an entity. Ground unit data is received from the federation, and each unit's data is stored as a row in the RDB's `units` table, thus allowing units to be queried in interesting ways. Since units have little behavior within JNEM itself, there is no Tcl object associated with each unit.

Fixed site entities, e.g., power plants and hospitals, do have significant behavior within JNEM, and hence have significant amounts of associated code and data. In JNEM v1, all entity data was stored in the RDB, and the related code for a particular entity type resided within one more singletons within the application, singletons whose primary task was something else. There were three advantages to this approach:

- Entity data could be easily queried.
- Entity data was trivially included in saved checkpoints.
- Entities by their nature are transient, and the housekeeping nuisance of saving and restoring transient instances was thereby avoided.

There were also disadvantages:

- Code related to an entity type was split between different modules, wherever the entities were used; there was no central place for it to live.
- All changes to entity data required explicit SQL updates to the RDB, thus making the code uglier and more verbose.

(A note on using an SQLite3 database for an application's record data: it is sometimes reasonable to write an API for updating records in the database, but it is rarely reasonable to write an API for querying records. Such an API is never as expressive as SQL SELECT statements, and it is *much* faster to do a SELECT and allow SQLite3 to iterate over the selected entity data than it is to do a `foreach` over entity IDs and then query the RDB for each entity's data.)

In JNEM v2, it soon became clear that the entity-related code was becoming unmanageably ugly, and that some entities could benefit from being represented as Tcl objects, i.e., as Snit instances. Now, saving and restoring transient instances is a nuisance: on a restore, one must destroy all existing transient instances, and then recreate the saved instances. There are two aspects to this problem: creating the restored objects with the correct names, so that they can be used by other code, and creating the restored objects with the correct data. Moreover, we wished to make this change *without* losing the benefits of storing entity data in the RDB. The result was the "databased-backed objects" pattern. The details of the pattern are as follows.

Each entity type is mapped to a particular RDB table; individual entities are represented as rows in the table. Each entity has a unique ID, which is the table's primary key. Entities are always addressed by their unique ID, not by any object name. Thus, the object names are arbitrary.

The first step is to make it possible to efficiently retrieve a Tcl object for a given entity, given only the entity's ID. At any given time, an entity might or might not have an associated Tcl object. Any routine that needs to access an entity as a Tcl object requests such an object, giving the entity's ID. The object is created, if it does not already exist, and is cached for later use. By convention, the routine requesting the object may only presume that the returned object name is valid until the routine itself returns. The routine may use the object name freely, and may pass it to other routines; however, only the entity ID should be saved in data structures. Entity object names are *not* persistent. If the state of the simulation changes, i.e., if state is restored from checkpoint, then the content of the RDB is presumed to be different. The cache of entity objects is cleared, and all such objects are destroyed. By this means, objects are created on demand and then retrieved as needed.

This part of the pattern is implemented by a singleton associated with the entity type; fixed site entities, for example, are managed

by the `site` singleton. The `site` singleton provides the following method (among others):

`site get` *`id`* `?-create?`
: Returns an object instance for the fixed site entity with the given *id*. Unless `-create` is specified, the entity must already exist in the RDB.

Thus, any routine that needs to access a fixed site's behavior will know the site's ID and will call `site get` to retrieve the instance. Any routine which merely wishes to query one or more fixed sites will query the RDB directly.

The entity objects are, perhaps surprisingly, not instances of the `site` type, but rather instances of the `siteType` type. As noted previously, defining a `snit::type` with both type and instance methods can lead to perplexing bugs if the type is called with a misspelled type method. (Note that `snit::widget`s are more or less immune to this problem, as a misspelled type method name is unlikely to look like a widget name.)

The second step is to tie an entity object's data to the entity's row in the RDB. When an entity object is created, it retrieves the entity's row from the RDB, and stores it in an instance variable, an array called `info`. This allows the object's methods to read this data without the cost of accessing the database. The entity object then provides at least the following methods:

`$entity set` *`name value`*
: Sets the value of entity field *name*, updating both the `info` array and the related RDB row.

`$entity get` *`name`*
: Retrieves the value of entity field *name* from the `info` array.

The rules that ensure consistency between the data stored in the RDB and the data mirrored in the entity objects are as follows:

- Only type methods of the entity singleton and methods of the entity instance type are allowed to use SQL queries to modify the contents of the entity type's table in the RDB.
- Other modules must use the entity singleton to affect entities as a group, and entity objects to update fields of individual entities. They may use the `set` method directly, or use other entity instance methods which call `set` indirectly.
- If the entity singleton updates existing entities, then the singleton's instance cache must be cleared. We will not try to update all existing entity instances immediately; instead, we will recreate them on demand.

This pattern preserves the advantages of saving entity data in the RDB while allowing entity code to be well-structured and easily maintained, at minimal cost in code complexity.

## 6. SQLite3 as a Memory Debugger

In conventional C or C++ programming, a perennial difficulty is finding out precisely what's going on in the program's memory, especially in environments where an external debugger is difficult or impossible to use. Tcl/Tk eases this problem by its very nature; with a small amount of work, any application can pop up a "console" window with a command line, so that the developer can query Tcl variables via Tcl commands. Even so, navigating complex data structures can still be tedious, especially when data is stored in Snit instance variables.

As noted above, JNEM uses its "Runtime Database", or RDB, as a structured memory store. The user is allowed to enter SQL "SELECT" queries for the RDB from the JNEM Console's command line interface; the query results are returned in tabular format. To the extent that the application's data is stored in the RDB, then, that data is easily browsed using arbitrary queries—which has the effect of turning SQLite3 into a memory debugger without peer. Any desired data or condition can be queried; and if desired, triggers can be used to trap particular conditions as they occur. It would be difficult to overstate the convenience of this feature.

## 7. The Scrollbar/Scrollable Pattern

The scrollbar/scrollable pattern is familiar to every Tk programmer; adding scrollbars to a text widget to create a simple text editor is a standard introductory example, but scrollbars can also scroll canvas widgets, frames, and listboxes. Scrollbar A and scrollable widget B have a symbiotic relationship such that if A's position is changed, B's viewport updates to match, and *vice versa*. At the lines of code level, A and B are wired together by a pair of callback/subcommand protocols:

- When A's position is changed, A's `-command` callback calls B's `yview` subcommand to update B's viewport.
- When B's viewport is changed, B's `-yscrollcommand` callback calls A's `set` subcommand.
- Both A and B are smart enough not to do anything if the subcommand tells them to do what they are already doing.
- A and B are kept in sync through all changes, whether triggered by the user or by the application.

The beautiful thing about this pair of callback/subcommand protocols is that it is defined by the arguments passed by the callback to the subcommand, not by either the callback or the subcommand name. This is the feature that allows the text widget to use two scrollbars at once, using the same protocols. Only the developer who glues A and B together needs to know the callback and subcommand names.

During JNEM v2 development, we realized that, properly abstracted, this pattern can apply to pairs of other kinds of widgets, as a "meta-pattern" if you will. For example, JNEM's log browser includes a search box called the "finder". Entering a search string in the box causes matching log entries to be highlighted in the body of the browser; further, the search box displays a count of the number of matches, and buttons to scroll the log browser from one match to another. If the content of the log browser is updated, then the search box must update its search.

Our initial implementations of the finder and log browser were rather ugly; we kept trying to make the log browser subordinate to the finder, and we kept having synchronization problems. The difficulty was that updates were received by both the finder and the log browser, and the log browser's updates were being mishandled, precisely because the finder was not involved. Pondering the Scrollbar/Scrollable pattern, we realized that neither the finder nor the log browser should be in charge; rather,

the two widgets must be peers, each keeping itself synchronized with the other. This could be called the Finder/Findable pattern; it has the same nature as the Scrollbar/Scrollable pattern. Given a finder A and a log browser B, A and B are wired together by a pair of callback/subcommand protocols:

- When the contents of A's search box is changed, whether by the user or by the application, A's `-findcmd` callback calls B's `find` subcommand, passing it the target string and search type (exact match, regular expression, etc.). B does the search and highlights the matches. Note that `find` is an ensemble subcommand with subcommands of its own.
- When B's search results change, B's `-foundcmd` callback calls A's `found` subcommand, passing it the number of matches found and the index of the match currently displayed. A updates its appearances, displaying the index and count, and enabling or disabling its arrow buttons appropriately.
- If matches were found, then A's `-findcmd` can call B's `find` command to navigate through the found matches.

JNEM has several "findable" widgets that will happily work together with the Finder widget.

The meta-pattern can be expressed as follows: given two objects, A and B, each of which must remain synchronized with the other in the face of updates to either one, glue the two objects together by means of a pair of callbacks and subcommands, such that A's callback calls B's subcommand, and vice-versa. Define the protocol in terms of the required arguments of each subcommand **only**, as it allows B to communicate with multiple A's at the same time, much as the text widget communicates with multiple scrollbars.

# 8. REFERENCES

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

[2] Tcl Manual, http://www.tcl.tk/man/tcl8.4/TclLib/Eval.htm.

[3] Roseman, Mark, "Ten Years of Rapid Development", 9th Tcl/Tk Conference, http://www.markroseman.com/pubs/tenyears.pdf.

[4] Duquette, William H., "Type Definition Objects", 12th Tcl/Tk Conference.

# 9. 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 Non-kinetic Effects Model (JNEM) for the U.S. Army Program Executive Office – Simulation, Training and Instrumentation (PEO STRI) in Orlando, Florida.