Poet: An OOP Extension to Tcl Supporting Constraints, Persistence, and End-User Modification

Philip J. Mercurio
Thyrd.org
mercurio@acm.org

Abstract

Poet (Prototype Object Extension for Tcl) extends the Tcl language with objects featuring dynamic, prototype-based inheritance, persistence, and one-way constraints between object attributes. Poet includes wrappers around the Tk and BWidget widgets that are automatically generated using introspection. This paper also describes **Poetics** (Poet Integrated Construction Set), a sub-project within Poet to create tools to allow a Poet application's code and user interface to be modified by the end-user, from within a running Poet application. The goal of Poetics is to provide some of the functionality of an integrated development environment to the user of a Poet application. An object inspector and code editor are the beginnings of the Poetics toolset.

Poet is an open-source project hosted at poet.sourceforge.net.

Keywords

Tcl/Tk, Object-Oriented Programming, Prototypes, Constraints, Persistence, End-User Programming.

Introduction

Application developers using dynamic, interpreted languages like Tcl/Tk [1] enjoy access to the command interpreter to inspect and modify the state of running programs, as an aid to testing and debugging, and for prototyping new code. They may also choose to make the command interpreter accessible to the user, to enable end-user modification of the program. This is more common if the application is itself a command-line program—most GUI applications do not expose the command interpreter to the end-user.

Poet (Prototype Object Extension for Tcl) is an attempt to support the development of GUI applications that can be modified by an end-user. We begin by constructing an object system that is fully dynamic and transparent and includes support for persistence. Poet also provides a constraint network that automatically maintains relationships between object attributes. Introspection is used to generate wrappers that define Poet objects for each of the Tk and BWidget widgets, allowing GUI objects to participate in the constraint network.

A set of tools known collectively as Poetics (Poet Integrated Construction Set) supports end-user inspection and modification of a running application. The primary tool is an object browser that displays an object's attributes (called *slots*), methods, constraints, and relationships to other objects. Poetics also employs Poet's optional type annotation feature so that appropriate content-specific editors can be created to edit slot values. A syntax-highlighting code editor allows an application's source code to be edited, saved, and reloaded in a running application.

As of this writing, the Poetics tools are incomplete and not ready for end-user use. However, even in its current, limited form Poetics can be very useful to the Poet application developer.

Related Work

The primary inspiration for Poet is Self [2], a dynamic object-oriented programming language and environment. Principal to the Self UI is its *liveness* [3], the sense that the user gets that the programming objects being manipulated are real and can be visibly changed. Self was also the inspiration behind using prototype-based inheritance rather than class-based in the design of Poet's object system.

Poet's constraint network is an implementation of one-way constraints, as used in ThingLab [4], Garnet [5], and Amulet [6]. Like these systems and in contrast to Self, Poet is intended as a platform for the development of desktop applications, not, primarily, as an interactive programming environment. Poetics' end-user programming features are not intended for the typical user of a Poet application, but for a subset of sophisticated user/developers ("gardeners" [7]).

Many other OOP extensions to Tcl exist [8], some class-based and some prototype-based. Poet sacrifices safety and some aspects of modularity in favor of a completely open and dynamic object model. All slots of a Poet object are accessible to be read or modified, and all of an object's methods can be invoked (or modified) from any code—there is no mechanism for limiting access. Message dispatch in Poet is completely dynamic, an object's response to a message may change from one invocation to the next due to changes to the object's ancestors. This freedom is embraced by several other Tcl OOP extensions, such as XOTcl [9] and Snit [10].

An extension called theObjects [11] released by Juergen Wagner in 1994 served as the basis for Poet. theObjects

implements prototype-based inheritance as a C extension to Tcl. In 1996 I ported it to Tcl7.5/Tk4.1, and in 1997 I began work on a complete redesign and port to Tcl8.0 under the name Poet. Some of the syntactical differences between theObjects and Poet are inspired by Object Tcl [12], another dynamic OOP extension to Tcl.

Although most of Poet is implemented in Tcl, it has always benefited from having a C core. Since the primordial Poet object, `Object`, is implemented in C, there was little overhead resulting from choosing to write a particular method in C vs. Tcl. Approximately one third of the methods on `Object` are implemented in C. Poet's constraint network is also implemented in C.

Objects, Methods, and Slots

Poet begins by defining the Tcl command `Object`, the ancestor to all Poet objects. A Poet object consists of a C structure, a Tcl command, and a set of Tcl arrays, called the object's *dimensions*, which hold most of the data pertaining to the object. Since the dimension arrays are accessible from both C and Tcl, much of Poet's internals can be implemented in Tcl. The C code for Poet is about 5000 lines, one quarter of which implements the constraint network. The remainder of the non-GUI portion of Poet is implemented in approximately 4300 lines of Tcl.

Objects are constructed by their parent and destruct themselves. Poet uses Tcl's autoloading support to load an object's source code file when the object is first referenced. The first command in an object's source file is an invocation of the `construct` method on its parent, which causes the parent to be autoloaded if it doesn't already exist. Multiple inheritance is supported by the method `mixin`, which also makes sure the new parent is loaded. Thus loading an object causes all of its ancestors to be loaded as well.

Poet does not yet implement garbage collection, it is expected that objects will be explicitly destroyed and will clean up after themselves by overriding the method `destruct`. In addition, one of the dimensions of an object is used to store a list of *goodbye* scripts, Tcl scripts that will be automatically invoked when the object destructs. Goodbye scripts are often used to notify another object about this object's demise. For example, a user interface object might attach a goodbye script to an object that it is displaying, to receive a notification when the object is deleted.

When a method is invoked on an object, the inheritance hierarchy, starting with the object itself then its parent and mixins, is searched for an implementation of the method. Internally, a Poet method is a Tcl procedure with the target object available as `$self`. Methods overriding an ancestor's implementation do not automatically invoke the overridden method, but any method can be called on any object using the method `as`. For example, an object that behaves just like Object but announces when it is destroyed would be implemented as in Example 1.

```
Object construct VerboseObject

VerboseObject method destruct {} {
    puts stderr "$self destructing"
    $self as [VerboseObject parent] \
          destruct
}
```

Example 1. Complete definition of an object that announces its destruction.

If the argument given to `Object construct` ends with "*", the name of the object constructed will consist of the argument followed by a serial number guaranteed to be unique in this interpreter—an anonymous object name. If "@" is used instead, a persistent version of the anonymous serial number is used, returning a name unique within the persistent storage and suitable for naming a persistent object.

Object attributes are accessed via the `slot` method. Given one argument, `slot` returns the value of that slot as seen from the current object. If the slot does not appear locally on the object, its ancestors are searched for a value. If it is not found, the null string is returned but no error is generated (the method `hasSlot` can be used to distinguish between an object that doesn't recognize a slot name and one that has the slot, but set to the null string).

With two arguments, `slot` sets the value of the named slot locally on `$self`, overriding any value set on an ancestor but not changing the ancestor's value for the slot. If the local version of the slot is later removed, the inherited version once again becomes available, as shown in Example 2.

Slots with names beginning with "_" (underscore) are private and are not subject to inheritance. If a private slot is referenced and there isn't a value set for it on `$self`, null is returned. Private slots are also not part of the persistent storage for an object. The underscore may also be used to indicate private methods, but this is just a naming convention and is not enforced by Poet. Any slot or method may be accessed on any object, regardless of whether it has a public or private name.

Slots may be designated as *active slots* that trigger a method invocation when they are read and/or written. A slot named `test1` on object `alpha`, activated for writing, will cause a method called `test1>` to be executed on `alpha` when the value of `test1` is changed. (The suffix indicating a read-active slot's method is "<".) The `test1>` method may veto the attempt to set the slot and set it back to an allowable value, acting as a guard. Note that the slot `test1` and the method `test1>` are subject to

inheritance separately, so that a slot's behavior and value may reside on different objects.

Slots are implemented as Tcl variables, they are elements of one of the object's dimension arrays. Tcl's trace mechanism is used to implement active slots and to trigger events in the constraint network.

```
% Object construct a
a
% a slot name "The Letter A"
The Letter A
% a slot name
The Letter A
% a construct b
b
% b slot name
The Letter A
% b slot name "The Letter B"
The Letter B
% b slot name
The Letter B
% a slot name
The Letter A
% b unslot name
% b slot name
The Letter A
```

Example 2. Sample dialog with the Tcl interpreter showing the creation and removal of a slot.

Persistence

Poet objects can be designated as persistent by having them mix in the object `Thing`. The object `ThingPool` is used to specify the storage for the pool and to load and save it. When the pool is written to storage, a Tcl script is created for each Thing object consisting of the commands that construct the object, set its public slots to their current values, and declare its parents, methods, etc. If the storage provided is a directory the scripts will be written as separate files, if a file is provided they will be written as a virtual file system using tcllib's VFS support. The current value of the @ anonymous name counter and an index to the persistent objects are also saved.

When a `ThingPool` is opened, the index and counter are read. Things are then autoloaded as they are referenced. If an object has component objects that it depends on it can override the method `Thing_postload` to cause them to be loaded when it is loaded, or it can just allow them to be autoloaded later. `Thing` also arranges for the persistent storage file to be deleted when the object destructs.

The code necessary to set up for persistence can be as simple as in Example 3. This code causes the first command line argument to the program to be opened as the persistent storage (either a directory or file) and made writable. We then redefine `exit` so that the pool will be closed when the application exits. All that remains is to mix `Thing` into any objects that should persist.

```
ThingPool setFile [lindex $::argv 0]
ThingPool slot writable 1
ThingPool open

rename exit crash
proc exit {{returnCode 0}} {
    ThingPool close
    crash $returnCode
}
```

Example 3. Setup for persistence.

Constraints

A slot's value can be constrained via the method `slotConstrain`, which takes the name of the slot as its argument. This initiates a search up the inheritance hierarchy for a formula with the same name as the slot. The formula is a Tcl script that is evaluated, its return value becomes the value of the slot. Like a method, a formula has access to the current object via `$self`. During the evaluation, accessing any other slot on any object causes it to be automatically recorded as a *source* in the constraint network, with the constrained slot as its *destination*. At any time in the future, when a source slot's value is changed, an event will be queued to recalculate each of its destination nodes.

Although Poet constraints are one-way, they can be arranged such that there is a circular dependency relationship between slots in the network, and it is possible to get caught in an infinite loop. The Poet distribution includes a demo of three scales ranging from 0 to 600, each of which has a formula constraining the slot containing its value to two times the value of the next scale, with the last scale dependent on the first. The formula also limits the value to 512, so that, in most cases, the network will stop updating once each scale reaches 512. If all three slots are constrained simultaneously, however, it is possible to get caught in a loop. To avoid this, each invocation of `slotConstrain` should be followed by a call to Tcl's `update` to allow the event queue to run and the constraint network to incorporate the new constraint.

A problem with automatically recording constraints when executing a formula is that incidental slot references not related to the computed value cause irrelevant dependencies to be created. Poet provides two means of dealing with this problem: 1) An object can be specified as the constraint limit, only objects descended from it are allowed to participate in the network. 2) Within a formula, a script can be executed as a side effect, so that no constraints are recorded, by passing it as the argument in the command

`Poet sideEffect` *`script`*.

Even if there are no circular or irrelevant dependencies in the network, it might take a noticeable while to settle into a stable state if there is a lot of computation to be done. A formula can indicate that it has not completed computing its value by returning with a special error code containing a unique token (such as the name of a Poet object). The error is caught and the token is associated with the destination slot. The slot is not assigned a value and another event to compute its value is queued. As the formula is invoked repeatedly, it uses the token to indicate when it is resuming work on the computation, and when it has completed. This mechanism can be used to maintain liveness when using a constraint network requiring significant computation or frequent UI updates.

Type Annotations

Poet slots, being Tcl variables, can hold any value. Poet provides a means of annotating a slot with an indication of what types of values the slot should contain. This is implemented by adding a *type* dimension to objects. The method `type` has the same usage as `slot`: when invoked with the name of a slot and an arbitrary string it associates the string with the slot name. If invoked with just a slot name, it returns the type string associated with that slot, searching the object's ancestors if necessary. Note that, as with formulas, a slot's type annotation may be inherited from a different object than the one that holds the slot's value.

Poet's type annotations do not comprise a type system in the traditional sense. Slot values are not required to conform to their types, and type inferencing is not used to validate expressions. In fact, the use of type annotations is completely optional. If the type of a slot is requested but none was declared, the null string is returned and no error is generated.

Poet does not use types to limit the allowable values for a slot, but the application programmer can chose to write objects whose slots conform to their types. The application programmer may also define their own type annotations with their own semantics, ignoring the type strings defined in the Poet library.

The Poet code declares types for many slots but does not use them until we get to Poetics, where they are used in the introspection of Poet objects. Poetics defines a set of type strings representing most of the types used in setting the options of Tk widgets (integers, reals, colors, font names, etc.). These types are used to select an appropriate widget to edit each slot of an object in the object browser. This is further discussed in the section on Poetics below.

ProtoWidget & Assimilation

Poet supports the construction of custom toplevel windows and widgets by providing a set of wrapper objects for the existing Tk and BWidget widgets. A preprocessing Tcl script assimilates a widget set by creating one of each widget type, using Tk's introspection to examine its options, and writing out a Poet object with a write-active slot for each option. When a slot is written the corresponding option is reconfigured on the enclosed Tk or BWidget widget. Types are also assigned for the slots, based on information gleaned from the Tk source code by a pre-preprocessing script. These type hints, represented in a table mapping widget names and attribute names to a type, have to be further edited by hand before they are ready to be used by the assimilation script.

Example 4 shows the code generated for the background option for a Tk button on the X11 platform (the method `primary` returns the enclosed button's Tk pathname). The last line in the example makes the slot active when written.

```
Tk_Button slot background #d9d9d9
Tk_Button method background> {value} {
    set p [$self primary]
    if {$p ne ""} {
        $p configure -background $value
    }
}

Tk_Button type background <color>
Tk_Button slotOn background >
```

Example 4. Auto-generated code assimilating the background option of a Tk button.

The code generated by the assimilation preprocessor will differ slightly from platform to platform, mostly in the default values for various options (the default button background under Windows, for example, is SystemButtonFace). Assimilation must be run once on each platform, but doesn't need to be rerun unless an assimilated widget's API changes. The Poet installation directory can contain the output from multiple assimilation runs, the correct set for the current platform is chosen at runtime.

All Poet GUI objects are descendant from `ProtoWidget`, which implements underlying support for the autogenerated code. ProtoWidgets contain a slot called `layout` which specifies the geometry manager and options to be used to lay out the widget. If a widget's layout begins with a "`-`" (dash), it is assumed to be a list of options to `pack`, otherwise the first word of the layout is the name of the geometry manager and the rest are the options. In this way a Poet widget's layout can be specified in the same command as the rest of its options, and can participate in the constraint network as well.

ProtoWidget adds additional arguments to its construct method so that the values of slots can be set using "`-slotname value`" argument pairs, making creating a widget in Poet cosmetically similar to Tk. Example 5 shows a complete Poet program that creates a window with

a scale ranging from -7 to 7, and a button labeled “Reset”. The button sets the value of the scale to 0. The state of the button is constrained so that it is disabled any time the scale is already 0. The window created by running Example 5 is shown in Figure 1.

```
package require Poet

Tk_Scale construct scl . \
    -from -7 -to 7 \
    -orient horizontal \
    -layout {-side top}

Tk_Button construct btn . \
    -text "Reset" \
    -layout {-side top} \
    -command "scl slot value 0"

btn formula state {
    expr {[scl slot value] == 0 ?
        "disabled" : "normal"}
}
btn slotConstrain state
```

Example 5. Complete Poet program demonstrating a constrained widget option.

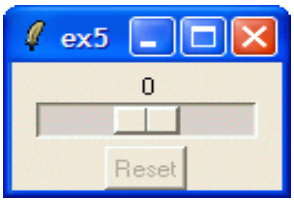

Figure 1: Output of Example 5

Assimilation actually generates two source files for each widget, the second one is mostly blank and will not be overwritten if it already exists. Additional methods to enhance the assimilated widget can be added here. The Poet library also includes a small set of custom widgets.

Poetics

Poetics is an attempt to support modification of a running Poet program's objects via direct manipulation. We begin by defining a set of type annotations oriented toward editing Tk widgets. A type string is of two forms. If it doesn't begin with a “<”, it is assumed to be the name of a Poet object, and the corresponding slot is expected to contain the name of an object that inherits from that object.

If the type string begins with “<”, it is a list, the first item of which is one of 18 known types enclosed in angle brackets (one slot of each type is displayed in the object browser shown in Figure 3). The remainder of the list, if present, contains parameters for this application of the type.

For example, a real number can be represented with the type annotation `<real>`, while an integer type string would be `<integer>`. Either of these two types can be extended with optional parameters specifying minimum, maximum, and step values. The type string

```
<real> -1.0 1.0 0.1
```

specifies a real value with a minimum value of -1 and a maximum of 1, and that it should be adjusted in increments of 0.1. The type string

```
<integer> 0
```

describes an integer with a minimum value of 0 but no stated maximum or step value. These type annotations are used by Poetics to configure a spinbox or scale widget for editing a slot of these types.

As another example, a slot with the type

```
<choice> alpha beta gamma
```

is allowed to take on only the value “alpha”, “beta”, or “gamma”. When creating an editor for such a slot, Poetics will use a combobox with the three alternatives present rather than a plain text entry widget.

The assimilated wrappers around the Tk and BWidget widgets have types like these defined for all of their slots, it is up to the Poet application developer to type the slots in the objects they write. When the object browser is viewing an object with typed slots, it presents an interface better suited for editing that object than one which edits all slots as string values.

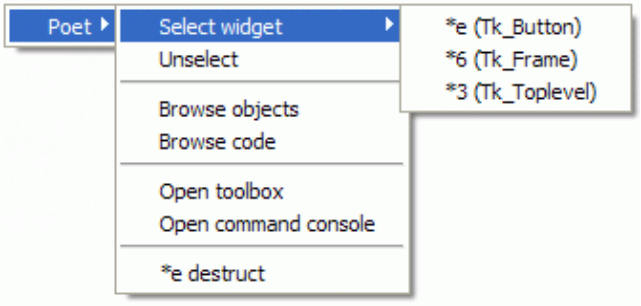

Figure 2: Poetics popup menu

Poetics also attaches a right-click popup menu to each ProtoWidget, similar to the one shown in Figure 2. The top item in the cascade menu to the right indicates that it was popped up over an object named `*e` whose parent is Tk_Button. The rest of the menu indicates that the button is contained within a Tk_Frame named `*6`, which is in a Tk_Toplevel named `*3`. In this manner any of the widgets under the location where the right-click occurs can be

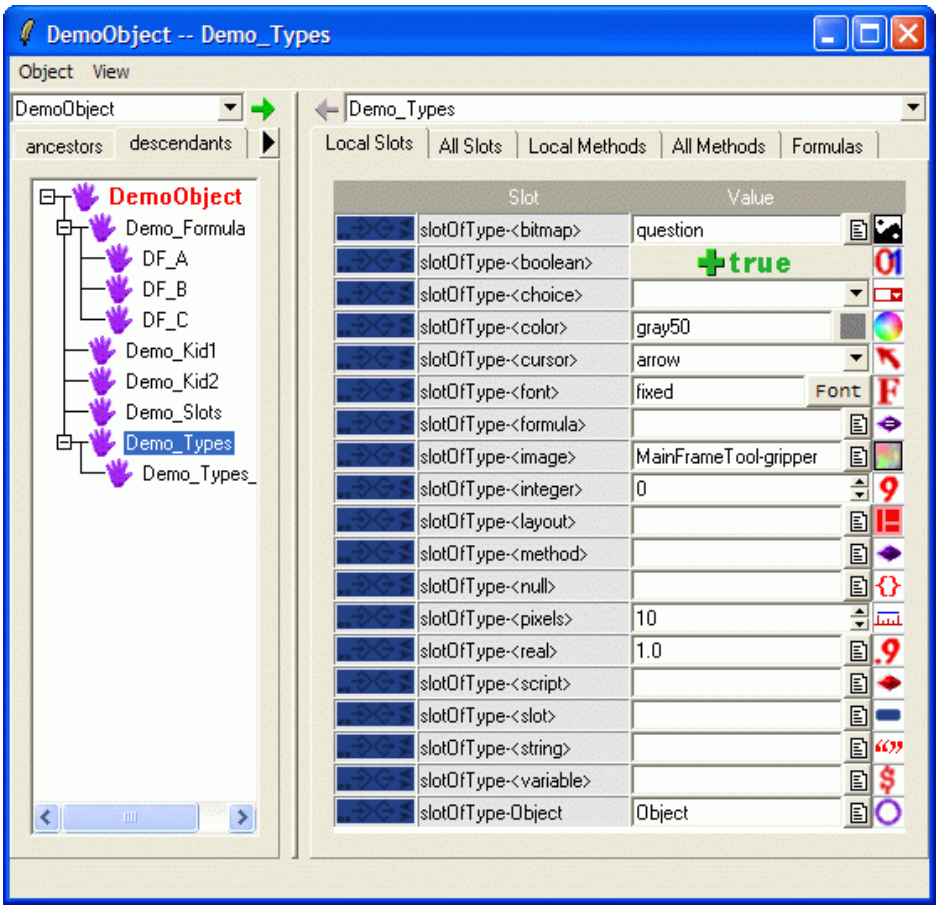

Figure 3: Object browser

selected.

Browsing and Editing Objects

The menu can be used to open an object browser, like the one shown in Figure 3. If a ProtoWidget is selected, the browser will open on that object. In this figure, the browser is displaying a set of objects created for demonstration purposes.

The browser consists of two panels, each with a combobox displaying a Poet object name. These comboboxes are also drop targets for icons representing objects, such as those displayed in the tree in the left panel. The icon indicates how the object name was generated: anonymous names are represented by a `*` icon, persistent anonymous names by a `@` icon, and non-anonymous names by a hand. The arrows next to the comboboxes control whether changing one entry changes the other to match. The browser is currently displaying two different objects, `DemoObject` on the left and a child, `Demo_Types`, on the right.

The left panel contains a notebook of tree widgets displaying various relationships between the selected object and other objects. The first two pages display the object's ancestors and descendants, DemoObject's descendant tree is shown. The next two pages are relevant only if the selected object is a ProtoWidget: one displays the widgets contained within the selected widget (such as the buttons contained within a frame), the other displays the containers or *shells* surrounding a widget. The last two pages apply to objects that have constraints on some of their slots and display other objects that have links into this object (are sources to formulas on this object) and those that have links out from this object (objects with formulas referencing slots on this object).

The panel on the right is a notebook of tables displaying various aspects of an object. Shown is a table of the slots local to the object Demo_Types. Each slot has an icon on the left that indicates if the slot is private or public, a source or destination (or both) in the constraint network, or active for reading or writing or both—this is shown better in Figure 4. The third column of the table contains an editor for the slot's value specific to the slot's type. For example, the editor for `slotOfType-<cursor>` is a combobox presenting the allowable names for a Tk cursor, while the editor for `slotOfType-<font>` is a text entry for the font name and a button that brings up a font selector dialog. This example shows the custom slot editors available as of

this writing, the default is a text entry widget and a button to open a multi-line text editor.

The next page of the notebook lists all of the slots available on the object, including those whose values are inherited. Inherited slots are highlighted in yellow, and clicking on an inherited value in an attempt to change it pops up a dialog for the user to confirm (or cancel) setting a new, local value for the slot.

Figure 4: Slot icons

Poetics includes a syntax-highlighting source code editor that can be opened via the ProtoWidget popup menu. The remaining three tabs of the object notebook display tables of the local methods, all methods, and all formulas defined for the selected object. Dragging a method or formula icon from the table onto a code editor will cause it to open the source file and highlight the method or formula definition. Currently, this only works for autoloaded objects, since Poetics relies on the autoload index to obtain the pathname of the source file given an object name.

Problems & Future Work

The dynamic aspects of Poet's object system make for a fluid programming experience, but they can also land the programmer in hot water. There is no protection for the contents of an object, so a slot can change value or a method can be redefined from any point in the source code. A common mistake is to copy the text of some methods from one object to another, and to accidentally end up with two definitions for the same method on the same object in two different source files. With autoloading enabled, a method may behave normally for part of the operation of the program, and suddenly change behavior when an unrelated object is loaded carrying an out-of-date version of the method. The procedure that creates the autoloading index attempts to detect problems of this sort, but it's still an easy trap to fall into. Poet trades safety for flexibility.

Poet's philosophy towards errors is to tolerate as much as possible. Attempting to obtain a nonexistent slot value is not an error, the null string is returned instead. Many errors are trapped by a dialog that allows the user to ignore the error (this dialog also has an option to drop into the tkcon [13] debugger, if available). The result is an environment that is, again, more programmer-friendly than safety-minded.

Poet, without Poetics, is fully functional and has been used for application development for the past two years. The Poetics tools are incomplete and unreliable, and are disabled by default when loading Poet. However, even in its primitive state, Poetics is useful to the Poet application developer for debugging and testing.

The first step towards an end-user usable Poetics is to implement editors for all of the known slot types. As can be seen in Figure 2, only a few slot types have custom editors, the rest have a button that opens a text editor. A particular challenge is found in the design of an editor for ProtoWidget layout options. Many good designs have been explored by Tcl IDEs like SpecTcl [14], Visual Tcl [15], and TkProE [16].

Poetics currently supports browsing and editing existing objects and code files, but does not support creating new ones. It is assumed that most of a Poet application's code would be written in the traditional manner, using a text editor, and stored as Tcl files. The objects appearing in the application's persistent storage would be primarily holders of data, with their methods belonging to ancestors defined in the application's source code. Mostly likely, a Poet application developer would not want these ancestral objects to be modified by the end-user and would make these source files read-only. Poet's installer does this when installing Poet's internal library.

To implement the creation of new objects in a running application, Poetics has a small toolbox window bound to the F7 key that, while currently limited in functionality, can become the locus for controls for creating objects. When the user creates a new object, the system then has to determine the disposition of the object, such as whether or not it should be persistent. If it is, is it then an addition to the Poet library, or is it part of the application's code, or is it part of the data saved by the application? I've begun investigations into subclassing Thing into subordinate objects that cause their descendants to be saved into different repositories, along with a table-based interface to manage multiple pools of persistent data. The pool used to store an object would depend on which child of Thing it mixes in, so that by default new objects would reside in the same pools as their parents. The persistence support currently available in Poet assumes one repository.

In addition to completing Poetics' support for the creation and direct manipulation of Poet objects, I'd also like to explore programming-by-demonstration [17] approaches to

creating Poet code. Straight-forward extensions of the existing tools will make it possible to create a new button in a running application and open a code editor on the button's command script. A more significant challenge would be augmenting the code editor so that it can monitor events occurring elsewhere in the interface, making it possible to demonstrate an action, such as changing the value of a slot, and have that represented as a command added at the current insertion point in the code editor. This would enable a mixture of hand-written and demonstrated code.

Distribution

Poet is an open-source community project released under the GNU Lesser General Public License and hosted at poet.sourceforge.net. It currently has been compiled and tested on the Windows and Linux platforms, compilation on other Tcl-supported platforms should be straight-forward. Included in the distribution is a starkit containing Poet and the versions of BWidgets, TkTable, TkHtml, and tkcon it depends on. When run standalone the starkit enables Poetics and opens a demo window, or the kit can be sourced to load Poet (with or without Poetics) into an existing Tcl interpreter.

Conclusion

This paper describes Poet, a OOP extension to Tcl that supports dynamic, prototype-based objects, persistence, and one-way constraints. The goal of the project is an environment where a sophisticated subset of an application's users can modify and extend it from within the application itself. The current, limited implementation of the Poetics end-user modification tools are valuable as debugging aids for the Poet application programmer.

Acknowledgments

Many thanks are owed for the excellent work and support provided by the authors of Tcl/Tk, BWidgets, TkTable, TkHtml, tkcon, and theObjects. Thanks are also due to Peter Kochevar for his insightful comments on a draft of this paper.

References

[1] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.

[2] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA'87*, 1987.

[3] S. Tanimoto. VIVA: A visual language for image processing, *J. Vis. Languages and Computing*, 127-139, June 1990.

[4] A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM Trans. Graph.* 5, 4 (Oct. 1986), 345-374.

[5] B. Myers et al. Garnet: Comprehensive support for graphical, highly interactive user interfaces, *Computer*, 71-85, Nov. 1990.

[6] B. Myers, R. McDaniel, R. Miller, B. Vander Zanden, D. Giuse, D. Kosbie and A. Mickish. The Prototype-Instance Object Systems in Amulet and Garnet. In *Prototype Based Programming*, James Noble, Antero Taivalsaari and Ivan Moore, eds., Springer-Verlag, 1999, pp. 141-176.

[7] M. Gantt and B. A. Nardi. Gardeners and gurus: patterns of cooperation among CAD users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (1992). ACM Press, New York, NY, pp. 107-117.

[8] Object-Oriented Programming in Tcl website: http://www.tcl.tk/about/oo.html

[9] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, pp. 163-174, Austin, Texas, USA, February 2000.

[10] W. Duquette. "Snit's Not Incr Tcl", http://www.wjduquette.com/snit

[11] J. Wagner. Tcl'ers Wiki entry on theObjects. http://wiki.tcl.tk/5681

[12] D. Wetherall and C. Lindblad. Extending Tcl for Dynamic Object-Oriented Programming. Proceedings of the Tcl/Tk Workshop 95, Toronto, Ontario, July 1995.

[13] J. Hobbs. Enhanced Tk Console: tkcon. http://tkcon.sourceforge.net

[14] SpecTcl http://spectcl.sourceforge.net

[15] Visual Tcl http://vtcl.sourceforge.net

[16] TkProE http://tkproe.sourceforge.net

[17] A. Cypher, ed. *Watch What I Do: Programming by Demonstration*, MIT Press, 1993.

Errata

This version differs slightly from the published version: in Example 1 the last line read "... as [$self parent] ...". That does work, but the version presented here is more correct. Specifically, an object descendant from the original version of VerboseObject would enter an infinite loop when destructed.