Cmdr

Andreas Kupries ActiveState Software Inc. 409 Granville Vancouver, BC CA
andreask@ActiveState.com

ABSTRACT

The `cmdr` framework is a set of 12 related Tcl packages for the easy specification of the interfaces of command line applications. This means the declaration of the set of commands provided by the application, and their parameters, be they options or positional inputs. At runtime the internals of the framework, guided by the chosen specification, handle the bulk of processing $::argv. This covers determining the requested command, mapping argument words to command parameters, and validating them. Additional features of the runtime are an integrated help system and interactive command line shells with basic command and argument completion.

1. INTRODUCTION

Following a short overview of the framework's history, the bulk of the paper describes its design and internal structure. The paper concludes with a dicussion of limitations, warts, and possible future directions to take.

Development of cmdr[3] (speak Commander) began as a rewrite of the `stackato` command line client's[1] because of various problems with the existing system, namely the use a global namespace for all options of all commands, and the hacks needed to support hierarchical commands.

Wanted was a system able to at least handle most of the tasks of command line processing automatically, like parsing the list of words, determining the requested command, separating named options from positional inputs, matching argument words to the parameter, etc.

Further wanted were facilities to ease the implementation of less common tasks, like validation of parameter values, transforming from external (string) to internal representations, performing checks across multiple parameters, etc.

Existing packages like getopt[10] and cmdline[8] were much too simple for all of the above, so development of a new package was begun. The TEPAM[8] package, which actually comes relatively near to the requirements, was not remembered at the time. It is also more geared towards the creation of procedures with more complex argument processing, and not the description of the entire command line syntax of applications.

A requirement going beyond the above, and initially more of a wish, was support for an interactive command line shell. This spawned the development of a CriTcl[4] binding[9] to the linenoise[2] C library which portably handles all the necessary low-level tasks regarding terminal access and control. Build on top of this `cmdr` not only supports command line shells in various places of the runtime, but also has command line completion.

This paper describes version 0.4 of the framework.

As a small motivating example see listing 1 which specifies a command line providing 3 commands for the management of command aliases. This is actually a slice of `stackato`'s interface, modified to fit.

While this example does not have the necessary backend procedures required to actually run the commands, it is enough to demonstrate the integrated help system. Listing 2 shows the output of `tclsh ./alias0.tcl help`.

The decoupling of command names from their implementations we see here makes it easy to re-arrange and re-label the user visible commands without having to touch any other part of the code.

Listing 3 for example shows how the specification would look like if a command hierarchy is prefered over a flat set of names.

The associated help is shown in listing 2, albeit in the shortest variant possible, the result of running the command `tclsh ./alias1.tcl help --list` [1].

[1] And a bit of manual breaking of lines normally done automatically, guided by the terminal width. The default of 80 did not fit here.

2. SPECIFICATION

The domain specific language demonstrated in the previous section is the first, and main, interface seen by a developer.

It actually consists of three separate languages, for the specification of the overall command hierarchy (see table 1), of the individual commands in that hierarchy (see table 2), and their parameters (see table 3).

2.1 General

The conceptual model underneath the command hierarchy is that of a tree.

The inner nodes of the tree represent command ensembles, here called "officer"s. Each officer knows one or more commands, and delegates actual execution to their respective specification, which may be another `officer`, or a `private`.

The leaf nodes of the tree represent the individual commands, here called "private"s. Each private is responsible for a single action, and knows how to perform it and the parameters used to configure that action at runtime.

The same model is graphically presented in the Entity-Relationship-Diagram 1 below.

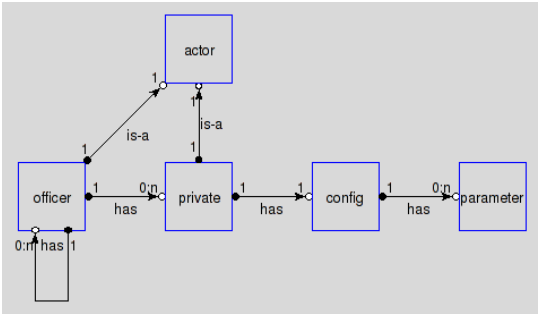

Figure 1: The Entities and their Relations

The not yet described "Actor" is the common base class for the ensembles and commands. The other, "Config", is the second interface seen by the developer, as the sole argument to the action callback argument of `private` (see 2nd-last row of table 1). This container holds all the declared parameters of the command the action is invoked for, and provides easy access to them through its methods at the time "Execution" (→ 3.4).

2.2 Officers

The three most common DSL commands used to specify officers (`officer`, `private`, and `description`, see table 1) have already been demonstrated in the examples (see listings 1 and 3).

alias *name* = *name'...*	Declare an alternate name for a command path.
alias *name*	Declare an alternate name for the previous command.
common *name script*	Declare shared code block.
default	Set last command as default.
description *text*	Set help text for the current ensemble.
ehandler *cmdprefix*	Execution Interposition.
officer *name script*	Declare a nested ensemble.
private *name script cmdprefix*	Declare a simple command. The script uses the commands found in table 2.
undocumented	Hide ensemble from help.

Table 1: Officer DSL

This leaves us with five commands not shown yet.

Of these `alias` is a structuring command, for the command hierarchy. It main uses are the creating of alternate command names, and of shortcuts through the command hierarchy. For example, `stackato`'s command specification for alias management is more like listing 3 and uses shortcuts similar to what is shown in listing 5 to provide the look of a flat namespace.

While `common` is a structuring command as well, its use is in structuring the specification itself. It creates named values, usually code blocks, which can be shared between specifications. Each block is visible in the current officer and its subordinates, but not to siblings. An example of such a block is shown in listing 6 (page), where it defines an option to access the subsystem for debug narative [8]. The example is actually special, as the code block named `*all*` is reserved by the framework. This code block, if defined, is automatically included at the front of all `private` specifications, i.e. shared across all specified commands. A very important trait for the option in the example, as it makes the debug setup available to all commands without having to explicitly include the block, and possibly forgetting such.

Use of the `undocumented` command influences the help generation, excluding all officers marked with it (and their subordinates) from the help. Note that subordinates reachable through aliases may be included, under the alias name, if not explicitly excluded themselves.

The last two commands influence the framework's behaviour at runtime

The `default` command sets up a default command to use at runtime if the currently processed word does not match any of the commands known to the officer. Without such a default an error would be thrown instead.

The `ehandler` command is possibly the most disruptive. It should normally only the specified at the top of the whole hierarchy. At runtime the framework will call the command prefix specified through it with a single argument, a script whose execution is equivalent to the phases "Parsing", "Comletion", and "Execution" of the framework, as described in section **??**. The handler must call this script, and can perform any application-specific actions before and after.

The handler's main uses are the capturing and handling of application-specific errors which should not abort the application, or shown as Tcl stacktrace, and for the cleanup of application-specific transient settings the parameter callbacks and/or command implementations may have set during their execution. This is especially important if the interactive command line shells of the framework are not disabled. Without such a handler and its bespoke cleanup code transient settings will leak between multiple commands run from such a shell, something which is definitely not wanted.

2.3 Privates

The specification of simple commands is relatively simple, with only seven commands (see table 2). The important parts are found in the parameter specifications, explained in the next section.

description *text*	Set help text for command.
interactive	Allow interactive shell.
undocumented	Hide command from help.
use *name*	Execute the named `common` block here.
input *name help script*	Declare a positional parameter. See table 3 for the available commands.
option *name help script*	Declare a named parameter. See table 3 for the available commands.
state *name help script*	Declare a hidden parameter. See table 3 for the available commands.

Table 2: Private DSL

The commands `description` and `undocumented` behave like the equivalents for officers.

`use` is the counterpart to `common` of officers, inserting the named code block it defines into the specification.

`interactive` influences the runtime. By default the only interactive command line shells are associated with the officers. Setting this marker activates such a shell for the command, to be invoked when required parameters do not have a value. The global command `cmdr::config interactive` can be used to globally activate this type of shell for all commands.

The remaining three commands all add one of the three kinds of parameters to the command.

2.4 Parameters

The parameters of `private` commands are the heart of the system, providing the space needed to transfer the command arguments to the implementations, and having the most attributes controlling their behaviour.

This complexity is mitigated strongly by the use of sensible defaults for each of the three possible kinds of parameter, i.e. positional `input`s, named `option`s, and `state` hidden from the command line. Each kind has its own construction command in the DSL for `private`, specifying the common information which cannot have defaults, i.e. the name identifying it to the system (→ 2.4.1, the help text describing it in informal speech, and, of course, the specification itself (see table 2).

The association between the specification commands, parameter attributes, and their defaults is found in table 3, with each group explained in the following sections.

2.4.1 Naming

We have two commands to influence the visible naming of all parameters.

As background, all parameters are named for proper identification within the framework and other Tcl code, i.e. the various calbacks, including a `private`'s action. This system name has to be unique within the `private` a parameter belongs to. Beyond that however the visible parameters have to be identified within help texts, and, in case of `option`s, for detection during "Parsing" (→ 3.2). That is the visible naming, seen by a user of any application whose command line processing is based on the `cmdr` framework.

The `label` command handles this, using the system name as default. Note that in most cases this default is good enough. The only use case seen so far is when two semantically equivalent `input` and `option` parameters clash, requiring different internal names due to the requirement for uniqueness, yet also the same visible name and flag within the help to highight their connection and equivalence.

In case of `option`s the `label` command and its default specifies the name of the primary flag recognized during "Parsing". If that is not enough for a specific `option` the command `alias` allows the specification of any number additional flags to be recognized. Note however that the framework automatically recognizes not only the specified flags, but also all unique prefixes, obviating the need for `alias` in many cases.

2.4.2 General control

The general handling of a parameter is influenced by three commands.

Like `officer`s and `private`s parameters can be hidden from the generated help. The command for this is `undocumented`, the same as for the first two . This is mainly of use to hide `option`s giving an application developer access to the internals of

Command	Attribute	Input	Option	State	Notes
–	name	–	–	–	Parameter name, unique within the `private`.
–	description	–	–	–	Help text.
–	visible	yes	yes	no	Parameter is (in)visible to "Parsing" (→ 3.2).
–	ordered	yes	no	n/a	Access during "Parsing" (→ 3.2) is ordered.
label *text*	label	`name`	`name`	n/a	Name to use in the help, and as primary flag (for an `option`).
alias *name*	aliases	n/a	none	n/a	Declare alternate flag for an `option` to be recognized by. Multiple aliases are allowed.
optional	optionality	no	n/a (yes)	n/a (no)	Declare `input` as optional.
test	acceptance	threshold	n/a	n/a	Control the matching of words to optional `inputs` (→ `??`).
undocumented	undocumented	no	no	n/a (yes)	Declare as hidden from help.
list	listness	no	no	no	Declare as list-valued.
default *value*	default	*	*	*	Set constant default value. Details in section 2.4.3.
generate *cmdprefix*	generate	*	*	*	Set callback returning the default value. Details in section 2.4.3.
interact *?prompt?*	interact, prompt	*	*	*	Enable the interactive entry of the string value. Default prompt derives from the `label`. Details in section 2.4.3.
defered	defered	no	no	yes	Defer calculation of the internal representation until demanded.
immediate	defered	yes	yes	no	Complement of `defered`. Calculate the internal representation during "Completion" (→ 3.3).
presence	presence	no	no	n/a	Declare as boolean `option` without argument. Implies `default` and `validate` settings.
validate *cmdprefix*	validate	*	*	*	Declare validation type. Details in section 2.4.4.
when-complete *cmdprefix*	when-complete	none	none	none	Set callback executed when the value becomes known.
when-set *cmdprefix*	when-set	none	none	none	Set callback executed when the string value becomes known.

Table 3: Parameters: DSL, Attributes, and Defaults

their application, something a regular has no need of, and doesn't have to know about.

Next is the possibility of specific `input`s not being required to be set by the user of the application, i.e. having sensible defaults (see also validation in section2.4.4). To mark such parameters use the command `optional`. During "Parsing" (→ 3.2) the system will then expend some effort to determine whether an argument word should be assigned to such a parameter, or not.

The `test` command is related to this, switching from the standard regime based on counting to a different one based on validation. The details are explained in section 3.2.

2.4.3 Representations

An important concept of parameters is something taken up from Tcl itself. The differentation between string and internal representations. Where Tcl uses internal representations to speed up its execution here this separation is that between the information delivered to the application by a user, and the application-specific data structures behind them.

All parameters will have an internal representation. This is usually derived from the string representation provided by the user. The details of that process are explained in section 2.4.4 about validation types. However we have cases where the user cannot specify a string representation (`state`s), or is allowed to choose not to (optional `input`s, `option`s). For these cases three specification commands are made available enabling us to programmatically choose the internal representation. They are `default`, `generate`, and `interact`.

The first, `default`, provides a constant value for the internal representation. The second, `generate`, provides a callback to compute the internal representation at runtime. This is useful if the default is something which cannot be captured as a fixed value, for example a handle to some resource, or a dynamically created object. These two commands exclude each other, i.e. only of them can be specified.

If none of them are specified, and we need a default (see the cases above) a default is chosen per the two rules below:

1. Use the empty string for a `list` parameter.
2. Use the default value supplied by the chosen validation type (See section 2.4.4).

The third command, `interact`, actually does not specify an internal representation, but activates another method for the user to specify a string value for the parameter, outside of the command line. As such it has priority over either `default` and `generate`, and can be specified with either. A parameter marked with it will interactively ask the user for a value if none was specified on the command line.

To recapitulate:

1. A string representation specified on the command line has the highest priority and goes through the chosen validation type to get the associated internal representation.
2. If activated via `interact` a small shell is run asking the user for a value (or more, as per `list`). The result goes through the chosen validation type to get the associated internal representation.
3. After that the internal representation is either the declared `default`, or the result of invoking the `generate` callback. As internal representations they are **not** run through the chosen validation type.

A command just noted in the recapitulation is `list`. It is used to mark parameters whose string and thus internal value should be treated as a list. This affects the handling of the parameter during "Parsing" (→ 3.2), by `interact` above, and the use of the validation type. The last two ask for multiple values, and feed the elements of the string value separately through validation instead of just the string value in one. During "Parsing" treatment of `option`s changes from keeping only the last assigned value to accumulation of all values. Similarly a list-`input` takes all remaining words on the command line for itself instead of just the current word. Because of this list-`input`s are only allowed as the last parameter of a `private`.

The last two specification commands dealing with the representations control when the internal representation is created. A `defered` parameter will do it on-demand, on the first access to its value. A `immediate` parameter on the other hand will do this during "Completion" (→ 3.3).

2.4.4 Validation

The answer to the necessity of moving between the string and internal representations described in the previous section are the validation types. Given a string representation they either return the associated internal representation or raise an error, signaling that the input was illegal. This part of their work, the verification of the legality of the input string gave them their name.

Because of the same necessity all parameters must have a validation type assigned to them, and the system will choose which, if the user did not. This choice is made per the six rules below and always returns one of the standard types shown in table 4.

1. Use "identity" if a `generate` callback is specified.
2. Use "boolean" if no `default` is specified and the parameter is an `option`.
3. Use "identity" if no `default` is specified and the parameter is an `input`.

4. Use “boolean” if the specified `default` value is a Tcl boolean.

5. Use “integer” if the specified `default` value is a Tcl integer.

6. Use “identity” as fallback of last resort.

Name	Complete-able	Accepts
boolean	yes	A Tcl boolean
identity	no	All strings
≡ pass		
≡ str		
integer	no	A Tcl integer
rdirectory	yes	A readable directory name
rfile	yes	A readable file name
rpath	yes	A readable path
rwdirectory	yes	A read/writable directory name
rwfile	yes	A read/writable file name
rwpath	yes	A read/writable path

Table 4: Builtin Validation Types

The general concept of validation types was taken from `snit` [7], and modified to suit `cmdr`. Where `snit`'s types expect only a single method to validate the input `cmdr` expects all types to support an ensemble of four methods, one for the basic validation and transformation of the input, another for the release of any internal representation so generated, plus delivery of a default representation and support for command line completion. The details (method names, signatures, etc.) can be found in table 5.

{∗}*vtype*...	Meaning
... complete *p x*	Return the list of legal string representations for the type and parameter *p* which have the incomplete word *x* as their prefix.
... default *p*	Return the default internal representation for the type and parameter *p*.
... release *p x*	Release the (resources associated with the) internal representation *x* of parameter *p*.
... validate *p x*	Verify that *x* is a legal string representation for *p,* and return the associated internal representation.

Table 5: Validation Type Methods

As an example the implementation of the standard boolean validation type is shown in listing 7. Note that while this example uses a `namespace ensemble` other methods are possible too, i.e. all the various object systems for Tcl would be suitable as well.

It should be noted that while `snit`'s validation types in principle allow for the transformation of input into a disparate internal representation, they never went so far as to allow complex representations which might require the release of resources after use.

Regarding the timing of a validation type's use of its methods see table 7 about the association of phases and callbacks.

The `validate` and `release` methods are primarily used during either “Completion” (→ 3.3) or “Execution” (→ 3.4), depending on the chosen deferal state. They may also be used during “Parsing” (→ 3.2), for optional `inputs` under the `test`-regime (→ 2.4.2).

The `complete` method will be used whereever the system activates an interactive command line shell where arguments may be assigned to parameters.

The `default` method on the other hand can expect to be invoked during “Dispatch” (→ 3.1), as part of the system's declaration processing, if not preempted by `default` and `generate` declarations for the parameter. Note here that the `default` method has the same signature as a `generate` callback and can be used as such. This is actually needed and useful when the default internal representation for a validation type cannot be expressed as a fixed value and its creation while parsing the specification itself is too early. We can still use the validation type for its generation, by hooking it explicitly into `generate` to change the timing of its invokation.

Not yet discussed so far is `presence`. This is best handled as part of the wider area of `option`s with a `boolean` validation type assigned to them. These have associated special behaviours, both in the handling of the specification, and during “Parsing” (→ 3.2).

First, normal `boolean option`s. They have automatic aliases declared for them, derived from their primary flag. An option named “foo” will have an alias of “no-foo”, and the reverse. During parsing the “foo” and “no-foo” flags have inverse semantics, and both are allowed to occur without option argument following the flag. This is in contrast to all other options which must have such an argument. The parser essentially uses the validation type to decide if the word after the flag is a proper boolean value, or not, i.e. an argument to assign to the parameter, or not.

Now `presence` declares a variant of the above, a `boolean option` without the automatic aliases, and never taking an argument during "Parsing" (→ 3.2). Its mere presence on the command line will set its parameter. Their `default` is consequently fixed to "false" as well.

2.4.5 Signaling

Of the four callbacks supported by parameters the first two, `generate` and `validate` have been described already, in the sections 2.4.3 about representations and 2.4.4 about validation types, respectively.

This section explains the commonalities between the callbacks in general, and the last two, for notifications about state changes in detail.

All callbacks are treated as command prefixes, not scripts. There are no placeholder substitutions, only arguments added to each command prefix on invokation. This does not harm the generality of the system, as complex scripts can be used via procedures or equivalents (i.e. `apply`).

The signatures expected by the system are shown in the tables 6 and 5.

Callback	Signature
generate *cmd*	{*}$cmd *param*
validate *cmd*	See table 5
when-complete *cmd*	{*}$cmd *param intrep*
when-set *cmd*	{*}$cmd *param string*

Table 6: Callback signatures

The last two callbacks not yet described are the two state-change callbacks defined via `when-set` and `when-complete`. They are invoked when either the string representation of their parameter is set (`when-set`), or their internal representation (`when-complete`). Through them the framework can actively drive parts of the application while processing the command line, whereas normally the application drives access to parameters through their methods (see table 9).

	Dispatch	Parsing	Completion	Execution
validate (default)	*			
validate (complete)		*	`immediate`	`defered`
when-set		*		
generate			`immediate`	`defered`
validate (validate)		`test`	`immediate`	`defered`
validate (release)		`test`	`immediate`	`defered`
when-complete			`immediate`	`defered`

Table 7: Execution Phases and Callbacks

Due to their nature these callbacks are invoked at runtime during either "Parsing" (→ 3.2), "Completion" (→ 3.3), or "Execution" (→ 3.4). The details are shown in table 7. The specification commands influencing the timing, i.e. forcing the use in a specific phase are shown in the intersection of callback and phase.

3. EXECUTION

At runtime a command line is processed in four distinct phases, "Dispatch" (→ 3.1), "Parsing" (→ 3.2), "Completion" (→ 3.3), and "Execution" (→ 3.4), explained in more detail in the following sections.

3.1 Dispatch

The first phase determines the `private` to use. To this end it processes words from the command line and uses them to navigate the tree of `officer`s until a `private` is reached.

Each word of the command line is treated as the name of the `officer` instance to descend into. An error will be thrown when encountering a name for which there is no known actor[2], and the current `officer` has no `default` declared for it.

On the converse, when reaching the end of the command line but not reaching a `private` the framework will not throw an error. It will start an interactive command line shell instead. This shell provides access to exactly the commands of the `officer` which was reached, plus two pseudo-commands to either exit this shell or gain help (`.exit,` and `.help).`

Execution of the command tree specification, i.e. generation of the associated internal `cmdr` data structures, is intertwined with this descend through the command tree. I.e. instead of processing the entire specification in full it is lazily unfolded on demand, ignoring all parts which are not needed. Note that the generated data structures are not destroyed after "Execution" (→ 3.4), but kept, avoiding the need to re-parse the parts of the specification already used at least once when an interactive command line shell is active.

[2] officer or private

3.2 Parsing

This is the most complex phase internally, having to assign the left-over words to the parameters of the chosen `private`, taking into account the kind of parameters, their requiredness, listness, and other attributes.

Generally processing the words from left to right `option`s are detected in all positions, through their flags (primary, aliases, and all unique prefixes), followed by their (string) value to assign.

When a word cannot be the flag for an option the positional `input`s are considered, in order of their declarations. For a mandatory `input` the word is simply assigned as its string value and processing continues with the next word, and the next `input`, if any. Operation becomes more complex when the `input` under consideration is `optional`. Now it is necessary to truly decide if the word should be assigned to this `input` or the following.

The standard method for this decision is to count words and compare to the count of mandatory `input`s left. If there are more words available than required to satisfiy all mandatory `input`s, then we can and do assign the current word to the optional input. Otherwise the current `input` is skipped and we consider the next. A set of condensed examples can be found in table 8, showing how a various numbers of argument words are assigned to a specific set of `input`s, `optional` and non. This is called the "threshold" algorithm.

Parameter	A?	B	C?	D?	E
#Required	2		1	1	
2 arguments:		a			b
3 arguments:	a	b			c
4 arguments:	a	b	c		d
5 arguments:	a	b	c	d	e

Table 8: Example: Mapping arguments to optional inputs by threshold

The non-triviality in the above description is in the phrase to "count words". We cannot simply count all words left on the command line. To get a proper count we have discard/ignore all words belonging to options. At this point the processor essentially works ahead, processing and removing all flags/options and their arguments from the command line before performing the comparison and making its decision.

The whole behaviour however can be changed via `test` (→ 2.4.2). Instead of counting words the current word is run through the validation type of the current `input`. On acceptance the value is assigned to it, otherwise that `input` is skipped and the next one put under consideration.

After all of the above the system will process any options found after the last word assigned to the last `input` to consider.

Errors are thrown if we either find more words than `input`s to assign to, or encountering an unknown option flag. Note that not having enough words for all required `input`s is not an error unless the framework is not allowed to start an interactive shell. In this shell all parameters are mapped to shell commands taking a single argument, the string value of parameter to assign. Additional five pseudo commands are available to either abort, or commit to the action, or gain help (`.ok, .run, .exit, .cancel,` and `.help).`

Parameters marked as `list`-valued also trigger special behaviours. For `option`s the assigned values get accumulated instead of each new value overwriting the last. For `input`s only one such parameter can exist, and will be the last of the `private`. The processor now takes all remaining words and assign them to this parameter. If the list is also optional then options may be processed ahead or not, depending on the chosen decision mode, as described for regular inputs above.

Then are the `boolean` and `presence` `option`s modifying the handling of flags and flag arguments. The details of this were already explained in section 2.4.4.

3.3 Completion

This phase is reached when all words of the command line have been processed and no error was thrown by the preceding phases. At this point we know the `private` to use, and its parameters may have a string representation.

All `immediate`-mode parameters are now given their internal representation. The parameters marked as `defered` are ignored here and will get theirs on first access by the backend.

This completion of parameters is done in their order of declaration within the enclosing `private`. Note that when parameters have dependencies between them, i.e. the calculation of their internal representation requires the internal representation of another parameter then this order may be violated as the requesting parameter triggers completion in the requested one on access. If this is behaviour not wanted then it is the responsibility of the user specifying the `private` to place the parameters into an order where all parameters access only previously completed parameters during their completion.

3.4 Execution

The last phase is also the most simple.

It only invokes the Tcl command prefix associated with the chosen `private`, providing it with the `config` instance holding the parameter information extracted from the command line as its single argument.

Listing 8 is an example of very simple action implementations, matching to the initial example specifications in listings 1 and 3.

All parameters declared for the `private` are made accessible through individual methods associated with each. A parameter named P is mapped to the method "@P", with all methods provided by the parameter class accessible as sub-methods. This

general access to all methods may be removed in the future, restricting actions and callbacks to a safe subset. A first approximation of such a set can be found in table 9.

cmdline	True if the parameter accessible on command line.
config ...	Access to the methods of the container the parameter belongs to, and thus all other parameters of the private.
default	Default value, if specified. See `hasdefault`.
defered	True if the internal representation is generated on demand.
description *?detail?*	Help text. May be generated.
documented	True if the parameter is not hidden from help.
forget	Squash the internal representation. See also `reset`.
generator	Command prefix to generate a default value (internal representation).
hasdefault	True if a default value was specified. See `default`.
help	Help structure describing the parameter.
interactive	True if the parameter allows interactive entry of its value.
is *type*	Check if the result of `type` matches the argument.
isbool	True if the parametr is a boolean option.
list	True if the parameter holds a list of values.
lock *reason*	For use in `when-set` callbacks. Allows parameters to exclude each other's use.
locker	The *reason* set by `lock`, if any.
name	The parameter's name.
options	List of option flags, if any.
ordered	True if the parameter is positional.
presence	True if the parameter is a boolean option not taking arguments.
primary *option*	True if the provided flag name is the primary option to be recognized (instead of an alias, or complement).
prompt	Text used as prompt during interactive entry.
required	True if the option is mandatory (`(input)` only).
reset	Squash internal and string representations. See also `forget`.
self	The parameter's instance command
set *value*	Programmatically set the string representation.
set?	True if the string representation was set.
string	Return the string representation, or error.
type	Parameter type. One of `input`, `option`, or `state`.
validator	Command prefix used to validate the string representation and transform it into the internal representation.
value	Return the internal representation. May calculate it now from the string representation.
when-complete	Command prefix invoked when the internal representation is set.
when-set	Command prefix invoked when the string representation is set.

Table 9: Parameter Methods

Calling "@P" without arguments is a shortcut for "@P value", i.e. the retrieval of the associated internal representation. Possibly calculating it if it is the first access and the parameter was in `defered` mode.

4. FUTURE DIRECTIONS

While I consider the framework to be quite mature, and it is already in use in a medium complex command line application, i.e. `stackato` [1], it is not without warts, nor opportunities for extension and improvement.

For example, the entire distinction between `immediate` and `defered` parameters is possibly not required.

The main use of `immediate` and "Completion" (→ 3.3) was to generate the internal representation of all parameters before invoking the action, validating all before use by the backend. Currently my thinking is beginning to doubt if that is as much of an advantage as I thought, over the converse, i.e. having everything `defered` and validating each parameter on first access. This would ignore all parameters not used by the chosen path through the code, leaving them unchecked. Which still offends my sensibilities somewhat.

Then there are the error messages currently generated for missing parameters, superfluous arguments, unknown options, etc. These are not as good as they could be, given the information available through the specification, i.e. all the help text. It might make sense to actually generate the help of the command and make it a part of the general error message.

Furthermore, currently all options are associated with specific commands. While the `common` and `use` commands allow the sharing of option definitions this is essentially duplication, even if hidden somewhat. Having actual global options not associated with any command, yet available to all is a request recently made for `stackato`. This would require some changes to "Dispatch" (→ 3.1), as it would have to recognize and process options as well, while navigating the command tree.

A wart I might have to live with is the hard-wired support for the special behaviours of `boolean` and `presence` options.

While I would like to replace that special code in "Parsing" (→ 3.2) with a general interface added to validation types and then moving them into the boolean type I currently do not see how this general interface should look like.

What is possible, would be to extend the validation types with optional methods to support type-specific interactive entry of values, to overide the simple shells currently used. These would then become fallbacks, used only for types without bespoke interaction. Related to that is the idea of adding a menu based interactor to the current set of general string and list entry interactors, to be used for validation types limited to a finite (and small) set of legal inputs. That is something which can be determined already, by inspecting the result of a type's `complete` method in response to an empty string.

A simple extension of the framework would the addition of more general validation types, i.e. useful to a broad class of applications. At least the builtin integer type could be extended to allow sub-typing, i.e. restriction to a range of integers instead of accepting all possible. Very application-specific validation types should be left out of the framework however.

On the side of the generated help the most important situations should be covered by the builtin formats, especially given that the `json` format provides access to essentially the entire specification in a portable format, convertible to any other format needed by a particular environment. Even so I am tempted to add direct support for at least doctools [8].

Structurally the help currently follows the hierarchy of commands and can show only either the single requested command, or all commands of a specific sub-tree. While this is ok for basic use a nicer help might have its own hierarchy, splitting the set of commands commands into logical sections which follow the use of the application instead. Support for this will require additional specification commands declaring the section(s) an application command is in.

APPENDIX

A. REFERENCES

[1] Various, Stackato Client https://github.com/ActiveState/stackato-cli

[2] Andreas Kupries, Linenoise. https://github.com/andreas-kupries/linenoise/,
fork of Steve Bennet, https://github.com/msteveb/linenoise/,
fork of Antirez, https://github.com/antirez/linenoise/

[3] Andreas Kupries, Cmdr. https://core.tcl.tk/akupries/cmdr/

[4] Andreas Kupries, Critcl https://github.com/andreas-kupries/critcl/

[5] Andreas Kupries, Kettle. https://core.tcl.tk/akupries/kettle/

[6] Andreas Kupries, Linenoise Utilities. https://core.tcl.tk/akupries/linenoise-utilities

[7] Will Duquette, Snit.
https://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/snit/snit.html#subsection11

[8] Various, Tcllib. https://core.tcl.tk/tcllib

[9] Andreas Kupries, Tcl Linenoise Binding. https://github.com/andreas-kupries/tcl-linenoise/

[10] Various, Tcl Core. https://core.tcl.tk/tcl/

B. LISTINGS

Listing 1: Simple alias management API

```
# -*- tcl -*
package require Tcl 8.5
package require cmdr
#package require foo-backend

cmdr create ::foo foo {
    private alias+ {
        description {
            Create a shortcut for a command (prefix).
        }
        input name {
            The name of the new shortcut.
        } {
            validate ::foo::backend::vt::notacommand
        }
        input command {
            The command (prefix) the name will map to.
        } {
            list
        }
    } ::foo::backend::alias::add

    private alias- {
        description {
            Remove a shortcut by name.
        }
        input name {
            The name of the shortcut to remove.
        } {
            validate ::foo::backend::vt::aliasname
        }
    } ::foo::backend::alias::remove

    private alias? {
        description {
            List the known aliases (shortcuts).
        }
    } ::foo::backend::alias::list
}

foo do {*}$argv
exit
```

Listing 2: Help for Listing 1

```
alias+ name command...
    Create a shortcut for a command (prefix).

    name    The name of the new shortcut.
    command The command (prefix) the name will map to.

alias- name
    Remove a shortcut by name.

    name The name of the shortcut to remove.

alias?
    List the known aliases (shortcuts).

help [OPTIONS] ?cmdname...?
    Retrieve help for a command or command set. Without
    arguments help for all commands is given. The default
    format is --full.

    --full  Activate full form of the help.
    --list  Activate list form of the help.
    --short Activate short form of the help.

    cmdname The entire command line, the name of the
            command to get help for. This can be several
            words.
```

Listing 3: Hierarchical commands

```
# -*- tcl -*
package require Tcl 8.5
package require cmdr
#package require foo-backend

cmdr create ::foo foo {
    officer alias {
        description {
            A collection of commands to manage
            user-specific shortcuts for command
            entry
        }

        private add {
            description {
                Create a shortcut for a command (prefix).
            }
            input name {
                The name of the new shortcut.
            } {
                validate ::foo::backend::vt::notacommand
            }
            input command {
                The command (prefix) the name will map to.
            } {
                list
            }
        } ::foo::backend::alias::add

        private remove {
            description {
                Remove a shortcut by name.
            }
            input name {
                The name of the shortcut to remove.
            } {
                validate ::foo::backend::vt::aliasname
            }
        } ::foo::backend::alias::remove

        private list {
            description {
                List the known aliases (shortcuts).
            }
        } ::foo::backend::alias::list
    }
}

foo do {*}$argv
exit
```

Listing 4: Help for Listing 3 (–list format)

```
alias add name command...
alias help [OPTIONS] ?cmdname...?
alias list
alias remove name
help [OPTIONS] ?cmdname...?
```

Listing 5: Specifying alternate names

```
alias alias+ = alias add
alias alias- = alias remove
alias alias? = alias list
```

Listing 6: Option shared by all commands

```
common *all* {
    option debug {
        Activate client internal tracing.
    } {
        undocumented
        list
        when-complete [lambda {p tags} {
            foreach t $tags { debug on $t }
        }]
    }
}
```

Listing 7: Validation type: Boolean

```
package require cmdr::validate::common

namespace eval ::cmdr::validate::boolean {
    namespace export default validate complete release
    namespace ensemble create

    namespace import ::cmdr::validate::common::fail
    namespace import ::cmdr::validate::common::complete-enum
}

proc ::cmdr::validate::boolean::release {p x} {
    # Simple internal representation. Nothing to release.
    return
}

proc ::cmdr::validate::boolean::default {p}  {
    return no
}

proc ::cmdr::validate::boolean::complete {p x} {
    # x is string representation. Result as well.
    return [complete-enum {
        yes no false true on off 0 1
    } 1 $x]
}

proc ::cmdr::validate::boolean::validate {p x} {
    # x is string representation. Result is internal representation.
    if {[string is boolean -strict $x]} {
        return $x
    }
    fail $p BOOLEAN "a boolean" $x
}
```

Listing 8: Simple alias management backend

```
# -*- tcl -*-
# # ## ### ##### ######## ############# ####################

namespace eval ::foo::backend::alias {
    namespace export list add remove
    namespace ensemble create
}

# # ## ### ##### ######## ############# ####################
## Command implementations.

proc ::foo::backend::alias::list {config} {
    set aliases [manager known]

    if {[$config @json]} {
        puts [jmap aliases $aliases]
        return
    }

    [table::do t {Alias Command} {
        foreach {name command} $aliases {
            $t add $name $command
        }
    } show display
    return
}

proc ::foo::backend::alias::add {config} {
    set name    [$config @name]
    set command [$config @command]

    manager add $name $command
    say [color green "Successfully aliased '$name' to '$command'"]
    return
}

proc ::foo::backend::alias::remove {config} {
    set name [$config @name]

    if {![manager has $name]} {
        err [color red "Unknown alias '$name'"]
    } else {
        manager remove $name
        say [color green "Successfully unaliased '$name'"]
    }
    return
}

# # ## ### ##### ######## ############# ####################
package provide foo::backend::alias 0
```