# ReacTcl - a lightweight framework for reactive programming in Tcl

Colin Macleod

colin_g_macleod@yahoo.com

CGM on Tcl wiki and chat

# Introduction

**Reactive Programming** is a technique where the programmer specifies a set of invariant relationships between the inputs and outputs of a system.

The system then reacts to changes in the inputs by automatically updating the outputs to maintain these relationships. Automatic tracking of the interdependencies of these computations allows them to be scheduled when required but not otherwise.

This can be viewed as a generalisation of the operation of spreadsheets to support more free-form and complex computations.

# Motivation

I started looking into this when I was working on financial trading screens where each user input could trigger many complicated updates to other elements on the same screen. I found that the existing procedural code to handle such updates was tricky, difficult to maintain, and sometimes inefficient.

In particular, adding new inputs which affect the existing calculations could cause the procedural code needed to perform the necessary updates to multiply in complexity. I have seen bugs introduced to production code by this process. Of course, one can be safe by simply recomputing everything on every update, but this may be unacceptable when some operations are slow, e.g. database accesses or calls to remote services.

# Example: inputs a,b,c; intermediates m,n,o; outputs x,y

*Dependencies*:

- m=M(a,b)  n=N(b,c)  o=O(c)  x=X(m,o) y=Y(n,o)

*Update Code*:

- If a changes, recompute m,x
- If b changes, recompute m,n,x,y
- If c changes, recompute n,o,x,y

# Motivation continued

In contrast, expressing the necessary computations declaratively was cleaner, safer, and sometimes even more efficient.  I believe this technique can be useful in a wide range of interactive applications.

That code was in C++, and was owned by my then-employer. It required templates to work multiple types of data, and an awkward system for tracking dependencies to make it thread-safe.  Fortunately neither of these is needed in Tcl, since *Everything Is A String*, and we never have multiple threads executing in one interpreter.

# Implementation

ReacTcl uses TclOO objects to represent reactive values.

A *Reactive* object represents a variable which can either be set to a specific value or computed from other Reactive objects. Such computed values are memoized and their dependencies on other Reactive values are automatically tracked so that these computations are re-run only when changes to their inputs have invalidated the currently memoized output value.

# Usage

- Create a Reactive object: react myVar
- Call the object with no arguments to get its value, or an error if that has not been defined: myVar
- Give a variable a fixed value: react myStr == {a b c}
- Give it an expr-computed value: react myNum = {3 * 2}
- Give it a chunk of code to evaluate:
  react myResult <- {string repeat [myStr] [myNum]}

# Live Demonstration

*see Appendix for log of demo*

The ReacTcl code is on the Wiki at:

https://wiki.tcl-lang.org/page/ReacTcl

# To Be Done

- Create a non-trivial example of when this approach can pay off.
- Mechanism to link input and output Reactive variables to the -textvariable of Tk widgets.
- Extension to arrays or dicts of values, memoizing individual elements.

# Appendix: Log of Live Demonstration

```
(reactcl) 52 % source reactcl.tcl
(reactcl) 53 %
(reactcl) 53 % react s
::s
(reactcl) 54 % s
::s is unset.
(reactcl) 55 % s == abcd
0
(reactcl) 56 % s
abcd
(reactcl) 57 % react n = {2 * 3}
::n
(reactcl) 58 % n
6
(reactcl) 59 % react r <- {string repeat [s] [n]}
::r
(reactcl) 60 % r
abcdabcdabcdabcdabcdabcd
(reactcl) 61 %
(reactcl) 61 % n = 5
(reactcl) 62 % r
abcdabcdabcdabcdabcd
(reactcl) 63 %
(reactcl) 63 % n <- {puts {Calc n}; expr { 1 + [m]}}  #Note
(reactcl) 64 %
(reactcl) 64 % r
Calc n
invalid command name "m"
(reactcl) 65 % r
invalid command name "m"
(reactcl) 66 %
```

```
(reactcl) 66 % react m = 3
::m
(reactcl) 67 % r
invalid command name "m"
(reactcl) 68 % react m
can't create object "m": command already exists with that name
(reactcl) 69 % n <- {puts {Calc n}; expr { 1 + [m]}}
(reactcl) 70 % r
Calc n
abcdabcdabcdabcd
(reactcl) 71 %
(reactcl) 71 % r
abcdabcdabcdabcd
(reactcl) 72 % m
3
(reactcl) 73 % m = 2
(reactcl) 74 % r
Calc n
abcdabcdabcd
(reactcl) 75 % r
abcdabcdabcd
(reactcl) 76 % s == xyzz
0
(reactcl) 77 % r
xyzzxyzzxyzz
(reactcl) 78 %
```

#Note: “puts {Calc n}” here lets us see when this calculation is re-run or when the previously memoized result is just reused.