Mighty Morphin' Widgets
A case-study in TclOO with Adaptive Widgets

Clif Flynt
Noumena Corporation,
8888 Black Pine Ln,
Whitmore Lake, MI 48189,
http://www.noucorp.com
clif at noucorp dot com

September 27, 2015

Abstract

The Adaptive Object Model is a relatively new concept in Object Oriented Design in which objects are configured during instantiation to match the runtime environment.

The TclOO object system's `mixin` facility provides the infrastructure for implementing and extending Adaptive Objects to reconfigure an object during use as the environment changes.

This paper describes the `fileWatch` widget as an example of a widget that redefines its methods to adapt to changes in the file it is displaying.

The `fileWatch` widget adapts to runtime conditions including

- A file being created or deleted.
- A file growing too large to display in main memory.
- A file being replaced by a gzipped version.

1 Introduction

Primitive widgets have limited adaptability. You can modify the bindings, background color, font, etc. for a button or a label, but it will still be a button or a label.

Web browsers use adaptive widgets to conform to the display environment: mobile, desktop, etc., but the functionality of the widget remains the same in all environments.

More complex widgets like the file selectors adapt to their environment in minor ways, displaying different sets of data in the file-selector window, allowing the user to specify paths, etc.

Even adaptable widgets like file selectors do not adapt when a folder is replaced by a zip file, or a remote server goes offline. Such conditions are commonly haneled by adding control code at the application layer.

The *Adaptive Object Model* as described at `http://adaptiveobjectmodel.com/` proposes objects that are customized during instantiation based on a set of rules.

TclOO extends this concept to objects that reconfigure themselves during use.

The `fileWidget` design arose from a need to display a file's contents despite the file being too large for main memory, newly created, modified or even gzipped. Responding to any of these changes needed to be done without control code in the application. The widget is built with the TclOO package and uses the `mixin` facility to reconfigure the widget during use.

The `fileWatch` widget is highly automated and adaptable. It can be attached to a non-existent file and will display the file when it is created. If the file grows too large for the usual scrolling text widget, it reconfigures itself to page in data from the disk. If the file is `gzipped`, the `fileWatch` widget automatically detects this and re-attaches itself.

The widget also contains optional search support and an optional basic file info display.

2 Introduction to Dynamic Object Oriented Design, TclOO Style

Traditional OO design paradigms are built around the concept that a class is a static structure defined early in a project's lifetime. It will encapsulate all the required information and behavior that an object of that class will ever require.

Like hard-coded variables, this is a concept that works well for a many of purposes, but doesn't truly reflect reality.

Tcl is a dynamic language. As such, TclOO supports modifying a class definition and object behavior in a dynamic manner–while an application is running. This feature of the language allows a Tcl script to adapt to a changing environment by redefining objects to match the current needs.

As an analogy, traditional OO design supports modeling a table saw or a drill press. TclOO supports modeling a Shopsmith (TM) that can be reconfigured as a table saw, router, drill press, lathe, to perform any required task.

2.1 TclOO Basics

TclOO is the set of Object Oriented commands supported by an extension to Tcl 8.5 and as part of the core in Tcl 8.6 and newer. It was developed by Donal

Fellows and borrows heavily from XOtcl (Gustav Neumann), incr Tcl (Michael McClennan), SNIT (William Duquette) and others.

The TclOO commands are defined in the `::oo` namespace to keep them safe from collisions with user-defined procedures.

A class is defined with the `oo::class create` command.

Syntax: `::oo::class create` *`name script`*

Define a class

name	The name of the class being defined
script	A Tcl script using TclOO commands which defines the new class

Most often a class definition includes a `constructor` to be invoked when an object is created, a `destructor` to be invoked when an object is destroyed and one or more methods.

```
::oo::class create fileClass {
  constructor {path} {
    variable State
    set State(channel) [open $path r]
  }
  method read {} {
    variable State
    return [read $State(channel)]
  }
  destructor {
    variable State
    close $State(channel)
  }
}
```

2.2 Hierarchies of related classes

The power of Object Oriented design comes from being able to combine small classes into larger classes with more functionality.

TclOO supports merging classes with two constructs, traditional inheritance and mixins.

The difference between inheritance and mixins is that an inherited class is commonly used for features that are expected to be permanent in the object while mixins are preferred for runtime modifications.

In Fantasy Role-Playing game terms, you might create a base class of `character`, and a derived class of `dwarf` that inherits all the methods from `character` and adds capabilities specific to all dwarves. When a dwarf object picks up a magic sword, the new capabilities are specific to this object and are not permanent. Thus the new capabilities should be added with a mixin class.

TclOO supports redefining both a parent class or a mixin class at runtime. In common use, the `super` class is not modified, though the mixin classes might be changed.

2.3 Traditional Class Hierarchies

The traditional method of creating class hierarchies is inheritance.

TclOO implements inheritance by allowing a class to declare a *super* class that it inherits methods and data structures from. Control can be passed from the derived class to the parent with the `next` command. Methods within an object are invoked with the `my` command.

```
::oo::class create fileTextClass {
  # Declare the parent class
  super fileClass

  constructor {path} {
    variable State

    # Create and map a text widget
    set State(textWidget) [text .t]
    grid $State(textWidget)

    # Pass control to the parent class constructor
    next $path
  }

  # Display the file in the text widget
  method display {} {
    variable State
    $State(textWidget) insert end [my read]
  }
}
```

A mixin can be declared with the class definition similar to declaring a parent class:

```
::oo::class create caesar1 {
  method read {} {

    # Call the parent 'read' method to retrieve the text.
    set d [next]

    # Trivial caesar cipher.
    foreach ch [split $d ""] {
      scan $ch %c x
```

```
      incr x
      append rtn [format %c $x]
    }
    return $rtn
  }
}

::oo::class create fileCaesar1TextClass {
  super fileTextClass
  mixin caesar1
}
```

2.4 Dynamic Class Hierarchies

Under the hood, TclOO is built around two commands, `oo::define` to define components of a class and `oo::objdefine` to define components of an object.

Rather than declaring a mixin with the class definition, a mixin can be added at runtime. From the previous examples, instead of defining the `fileEncryptTextClass` class with the `mixin` command, the code could add the encryption support at runtime:

```
fileTextClass create cipherShow /tmp/textmsg.txt
oo::objdefine cipherShow mixin caesar1
cipherShow display
```

The obvious advantage of this technique is that a new encryption policy can be created by defining a new class with a new `read` method and mixing that into a `fileTextClass` object when it's needed.

To further extend dynamic support, the `oo::objdefine` command can be invoked inside an object method allowing the object to redefine itself when conditions change.

The object hierarchy is modified using the `oo:objdefine` command.

Syntax: `oo::objdefine` *`objectName script`*
`oo::objdefine` *`objectName ?subcommand ?arg1 arg2 ...?`*
Define a feature of an individual object.

`className`	The name of the object to be modified.
`script`	A script with commands to modify the object. This may modify several features.
`subcommand`	A subcommand that defines a single feature to be modified.
`?arg1 arg2 ...?`	The arguments that a subcommand requires.

In the next code snippet, the adaptiveTextClass object will mix in the caesar1 class if the target file ends in a .crp. If the target file does not end in .crp, it's a text file and the base class read method is used.

```
::oo::class create adaptiveTextClass {
  # Declare the parent class
  super fileClass

  constructor {path} {
    variable State

    # Create and map a text widget
    set State(textWidget) [text .t]
    grid $State(textWidget)

    if {[string first ".crp" $path] > 0} {
      oo::objdefine [self] mixin caesar1
    }

    # Pass control to the parent class constructor
    next $path
  }
```

3 fileWatch widget

The fileWatch widget is composed of a master frame that contains a text widget and associated scrollbars and optionally file information (path and size) and forward/backward search frames.

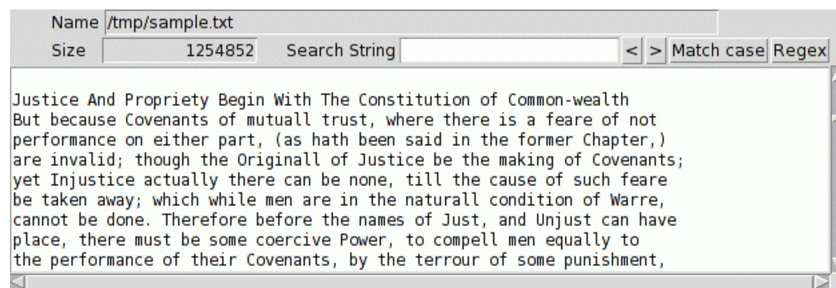

Figure 1: fileWatch widget

The files displayed by the fileWatch widget may be changing during runtime or too large to simply load and display or zipped.

The different conditions could be handled with a set individual procedures to handle the each type of file.

For example, a small file can read into a text widget with a scrollbar attached in the usual manner:

```
text .t -yscrollcommand ".ysb set"
scrollbar .ysb -orient vertical -command ".t yview"
```

The text widget becomes unwieldy when the file size exceeds a few megabytes. In that case, instead of invoking `.t yview` the command associated with the scrollbar will be some variant of `.t pagedYview`. The `pagedYview` procedure will calculate an offset into the file based on the arguments sent by the scrollbar, seek to the appropriate location in the target file and display a screenfull of the contents.

If a file is zipped it can be read by pushing the `gunzip` translator onto the channel with the code shown below (courtesy of Andreas Kupries' and Jacob Levy's work to add filters to the Tcl channel).

```
zlib push gunzip $State(chan)
```

This allows a channel to read zipped data, but the `seek` command is not supported for a zipped file.

The seek functionality can be implemented by closing and reopening a file then reading to the seek location. This is not as fast as a true seek, but the slow functionality is better than no functionality.

In the case of a zipped file, a private seek procedure needs to be added into the package.

Such a collection of procedures would implement all the required features for a single widget.

An application may require several `fileWatch` widgets in simultaneous use. Tracking private data for each widget using one or more global arrays can be done, but is not elegant. A namespace can be used to hold the private data, but the need to reconfigure the widget during operation makes that pattern cumbersome as well.

TclOO's support for dynamic object design is the clean method for creating this functionality.

3.1 Evolution of a design

The fileWatch widget grew from a need to display a file's contents.

The initial code was a trivial procedure to create a text widget with a scrollbar to display the data from a file. If the file was too large for a text widget, the first few kilobytes of the file were displayed followed by a message that "the file is too large."

The need to display very large files (multiple Gigabytes) drove the need to connect the scrollbar to a paging procedure that can load small sections of the file into the text widget.

```
method read {} {
  if {[file size $State(path)] > $State(maxSize)} {
```

```
    # assign scrollbar command option to paging method
  } else {
    # read all data and attach scrollbar to text widget
  }
```

A text widget that contains all the text in a file has no state connected with it beyond that held in the scrollbar and text widget. A scrollbar that's associated with a paging procedure has a great deal of state - the current seek location in the file, the open channel descriptor, etc.

A single widget can maintain the state in a global variable. However, multiple widgets need individual state variables. While this can be done with global arrays that borrow the pattern used by the `http::` commands, the problem is better solved with an object.

This led to collecting the code that handled opening, displaying, etc into a simple class with no hierarchy to centralize the associated methods and data.

The next requirement was that the users want to scroll to the bottom of a changing file and continue to see the last N lines–the behavior of `tail -f`.

This led to tweaks to the scrollbar methods.

When large files are finally closed, the parent application gzipped them to save space. This required that the widget know when a file was renamed so that it could be reopened with gunzip pushed onto the channel.

At this point, the self-contained class became unwieldy. The internal read and yview methods were getting too many layers of conditionals to cover small files, large files, small zipped files, large zipped files, files that were still changing, etc.

Moving the special case handling to separate modules is the obvious solution to the problem of excessive indents.

A traditional OO design using inheritance would solve the problem of handling different types of files, but does not address the issue of files that change from small to large or uncompressed to compressed at runtime.

TclOO's mixin functionality solves this problem.

3.2 Design

The `fileWatch` widget is composed of four classes, the primary class, `fileWatch` and the mixins for small, large and large zipped files.

A `checkFile` method is invoked at intervals to examine the attached file and determine if:

- the file has changed size
- a file that did not exist has appeared
- a file that did exist has disappeared
- the file has been gzipped

When one of these conditions exists, a new mixin may be merged into the object to adapt to the new conditions, replacing methods from previous mixins.

The hierarchy and methods are shown in the diagram below.

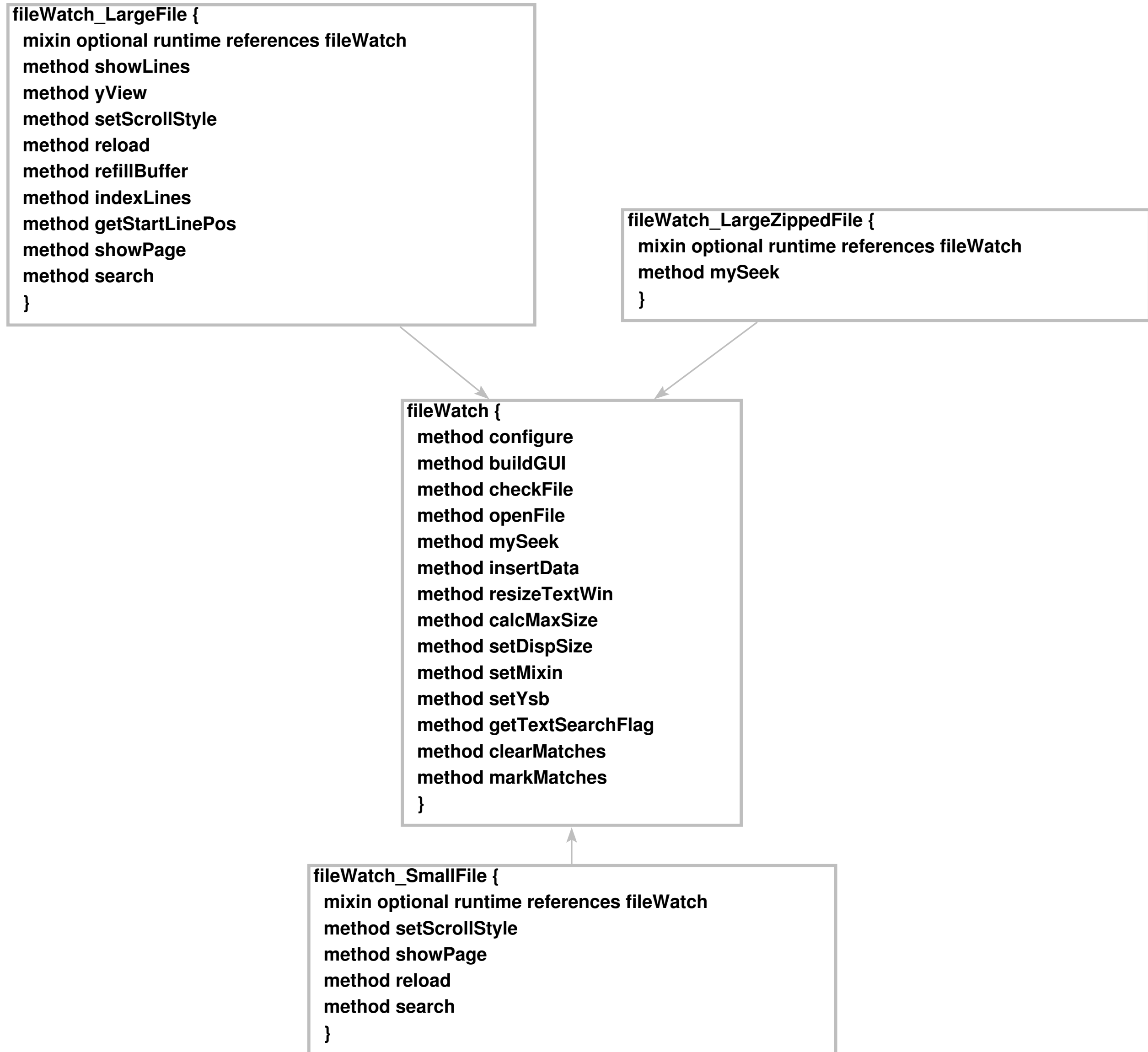

Figure 2: FileWatch Class Hierarchy

The critical methods are `checkFile` and `setMixin`. The actual code is fairly straightforward, but these two procedures implement the adaptive nature of the widget.

The `checkFile` method

- checks to confirm that a file exists
  - if the file does not exist, it's marked as such
  - if the file does not exist, but a gzipped version does exist, the .gz file is opened.
  - if the file does exist, it's opened
- checks the file size and calls `setMixin` if the size has changed.

The code for this method is a bit longer.

```
################################################################
#   method checkFile { {reset {} } }--
#    check for changes in a file's characteristics
# Arguments
#   NONE
#
# Results
#   setMixin may be invoked to modify object
#   New file channel may be created

  method checkFile {} } {
      variable State

    # If the file doesn't exist clean up if it used to exist.
    # If it exists as a .gz file, attach to that.

    if {![file exists $State(path)]} {
      # Did it used to exist?
      if {$State(size) > 0} {
        if {[file exists $State(path).gz]} {
          set State(path) $State(path).gz
          set State(displaypath) $State(path)
          set State(size) [file size $State(path)]
          # Can get here if file doesn't exist but file.gz does
          # In that case, chan == -1
          catch {close $State(chan)}
          my openFile
          my setMixin
        }
      }
    }
```

```
  # If it didn't exist, but does now, connect to the file

  if {($State(chan) == -1) && [file exists $State(path)]} {
    # File has appeared!
    my openFile
    my setMixin
    my setDispSize
    my reload
  }

  catch {file size $State(path)} newSize

  # If file size has changed save the new file size
  # possibly change mixins,
  # update the display if necessary.
  # Evaluate registered callbacks

  if {$newSize != $State(size)} {
    set State(size) $newSize
    my setMixin
    my setDispSize
    my reload

    if {$State(callback) ne ""} {
      if {[catch $State(callback) rtn]} {
        if {[file exist $State(path)]} {
          tk_messageBox -type ok \
          -message "FAILED: $State(callback)\n $rtn"
        }
      }
    }
  }

  # I'll be back.
  # Use catch in case the widget was destroyed
  # before the after event fires

  set State(afterID) \
      [after $State(interval) [list catch "[self] checkFile"]]
}
```

The `checkFile` method is triggered by an after event. Only public methods can be invoked from outside the class, so what is properly a private method is named with a lower case letter. The `ooutil` package contains wrappers to allow the method to be a private method, but in the interest of making the file-

Watch widget completely self-contained, I chose to use let checkFile be a public method.

The `checkFile` method examines the file but does not modify the object hierarchy. If the `checkFile` method finds any changes in the file, it invokes the `setMixin` method to determine if the object hierarchy needs to be modified.

Again the logic is a set of simple tests - is the file small enough to simply load into the text widget or does it require paging?

The lines that modify the object resemble this:

```
oo::objdefine [self] mixin fileWatch_LargeFile
```

```
################################################################
#   method setMixin {}--
#    Load a mixin for this file
# Arguments
#   NONE
#
# Results
#   Object hierarchy may be modified
#
  method setMixin {} {
    variable State

    if {$State(size) <= 0} {
      if {[catch {file size $State(path)} State(size)]} {
        return
      }
    }

    set currMixin [string trim [info object mixins [self]] :]
    set newMixin 0

    # Use paging if file is larger than 2M
    if {[string is double $State(size)] && $State(size) > 2000000} {
      # Large zipped files get special processing
      if {$State(zipped)} {
        if {$currMixin ne "fileWatch_LargeZippedFile"} {
          oo::objdefine [self] {
              mixin fileWatch_LargeZippedFile
              mixin -append fileWatch_LargeFile
              set newMixin 1
          }
        }
      } else {
        # File is plaintext, but large
        if {$currMixin ne "fileWatch_LargeFile"} {
```

```
            oo::objdefine [self] mixin fileWatch_LargeFile
            set newMixin 1
        }
          if {$newMixin} {
            # If we've changed mixin style, update display
            lassign [$State(ysb) get] start end
            if {$end < .99} {
              set charCount [string length [$State(textWin) get 0.0 end]]
              set pos [expr  $charCount * $start]
              set offset [expr {$pos/$State(size)}]
            } else {
              set offset 1.0
            }
            my yView moveto $offset
          }
        }
      } else {
         # File is small, load it and display
         if {$currMixin ne "fileWatch_SmallFile"} {
            oo::objdefine [self] mixin fileWatch_SmallFile
         }
      }
      my setScrollStyle

    }
```

The hierarchy does not include the object variable `State` that is unique to each object. Not surprisingly, the `State` variable holds information about the object's State. The table below shows the currently supported indices, organized by use. Not all files will have all the indices populated. For example, a small file does not use any of the large file paging parameters.

I used an array variable instead of independent variables because of the number of values required to define the state of the widget and the file. Using individual variables would have become awkward and difficult to maintain.

State Array

File information

State(path)	Actual file path
State(displaypath)	File path to display
State(dir)	Parent folder of file being viewed
State(chan)	File channel
State(type)	Type of item being watch - currently "file" or "NoExist"
State(row)	Row for widget if it grids itself in parent
State(col)	Column for widget if it grids itself in parent
State(zipped)	Set if file is zipped.
State(callback)	Optional Callback to invoke when file size changes
State(size)	Current size of file

Large file support

State(seek)	Offset into file
State(dataStart)	Offset of internal buffer into file for large files
State(displayStart)	Offset into internal buffer for display
State(lineIndex)	Index of line start offsets into buffer for large files
State(maxSize)	Calculated max number of chars in text widget.
State(lineCount)	Number of lines in text widget

Search support

State(regexp)	Use regular expression rules for search
State(matchCase)	Case match for search
State(searchStart)	Start location for next search
State(searchVal)	Value to search for

Window values

State(parent)	Window parent of widget
State(enableInfo)	Enable or disable the path/size display
State(enableSearch)	Enable or disable file search
State(DispSize)	Pretty formated file size
State(formatCmd)	Format command for pretty printing file sizes
State(infoArgs)	Optional key/value pairs to customize filename/size
State(textWin)	Window path to text widget
State(textArgs)	Arguments for text widget
	-wrap word -height 10 -xscrollcommand .delta1.xsb set
State(xsb)	X scrollbar window path
State(ysb)	Y scrollbar window path

File check interval

State(afterID)	Internally used for checking if file changes
State(interval)	File check interval - defaults to 1000

One of the goals for the `fileWatch` widget is to be a generic replacement for several custom widgets. As such it required the ability to be customized.

The following arguments are supported by the `fileWatch` constructor:

Creation time options

-path	Path to file
-displaypath	File name to display if enableInfo enabled
-textArgs	Args for text widget
-infoArgs	Args for information widgets
-addWin	Win-cmd
-parent	Parent frame
-interval	Refresh interval
-row	Top row for elements of the fileWatch widget
-col	Left column for elements of the fileWatch widget
-callback	Script to invoke when filesize changes
-enableSearch	1/0, 1 to enable the search elements
-enableInfo	1/0, 1 to enable the information elements
-formatCmd	Formatting for size or other information

The `-textArgs` and `-infoArgs` keys allow the application to customize the appearance and behavior of the text, information and search widgets. The argument is a list of key/value pairs.

The `-addWin` argument accepts a Tcl script to be evaluated. Combined with the `-row` and `-col` keys it lets the application add new windows or buttons to the `fileWatch` widget.

A pure default `fileWatch` widget can be created with a script like this:

```
fileWatch new -parent .default -path "/tmp/tstfile.txt"
grid .default
```

which creates a widget resembling this

Figure 3: Uncustomized fileWatch

You can customize the `fileWatch` widget by modifying the appearance of the text widget and info labels, adding a top label and a new button to archive the displayed file with this script:

```
fileWatch new -parent .customized -path "/tmp/tstfile.txt" \
```

```
-addWin {{button [my configure -infoFrame].b \
-text Archive -command archive} \
-textArgs {-font {arial 12 bold} \
-wrap word -height 10} \
-infoArgs {-font {courier 14} \
-relief solid -borderwidth 2} \
-maxSize 2 -enableInfo 1  -row 2 -enableSearch 1]]


grid .customized -row 3 -column 1 -sticky news
label .customized.lbl -text "Customized fileWatch" \
  -font {arial 16 bold}
grid .customized.lbl  -row 0 -column 0 -columnspan 2
```

which will generate a widget that resembles this:

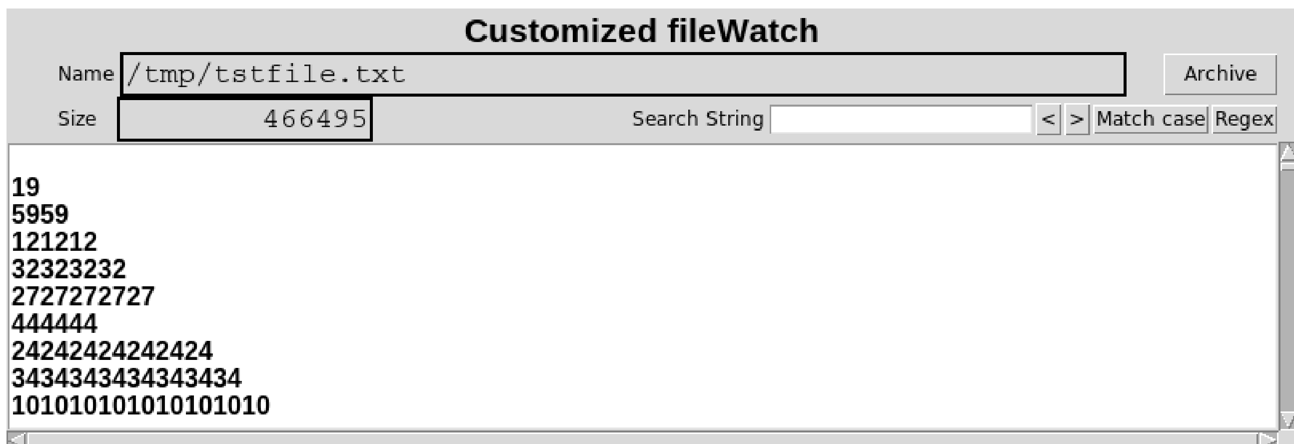

Figure 4: Customized fileWatch

4 Conclusion

The `fileWatch` widget demonstrates the power of the TclOO `mixin` construct to create widgets that respond to changes in their environment and reconfigure themselves to match.

The Adaptive Object Model is a new design pattern that has great potential for handling real-world situations ranging from financial applications to biology. Any design element that can change states may be best modeled using this technique.

As demonstrated with the `fileWatch` widget, the TclOO `mixin` command is well suited to developing systems using this model.