



# Clay

## A Minimalist Toolkit for Sculpting TclOO

Presented at

The 25th Annual Tcl Developer's Conference  
(Tcl'2018)

Houston, TX

October 15-19, 2018

### **Abstract:**

Clay is an attempt to take a career's worth of decent design ideas and condense them into a single language tool. Clay provides a vocabulary to describe the complex interactions possible in TclOO. It also allows programmers to harness that complexity in a deterministic fashion.

**Sean Deely Woods**

*Senior Developer*

*Test and Evaluation Solutions, LLC*

*400 Holiday Court*

*Suite 204*

*Warrenton, VA 20185*

*Email: [yoda@etoyoc.com](mailto:yoda@etoyoc.com)*

*Website: <http://www.etoyoc.com>*

## Introduction

I build and maintain several modules in Tcllib. For my work in Tcllib I've often wished for a robust framework to call on when building libraries of TclOO code. I have discussed frameworks at other conferences, notably Tool and Tao/Tk. But they are an ill fit to this task because they are principally designed to make megawidgets. Their concepts of event handling is of no help at all to develop a sockets application. They also introduced a such a number and variety of methods into their metaclasses that they are not a good fit for a general purpose library.

I had gotten around this problem with Practcl by simply making a standalone meta class. While developing the Httpd module, I found myself re-creating many of those same features. At first I adapted Tool to the job. But later found myself stripping that out and starting over with a fresh metaclass.

Rather than simply write another one-off meta class, I figured it was about time to write a framework for non-tk applications.

Both Practcl and Httpd makes heavy use of mixins. In Practcl this allows build products to take on behaviors via configuration options. In Httpd mixins are used in dispatching requests. Mixins also allow developers to modify the server. All of these interactions rely on begin able to answer questions about classes, before actually mixing them into an object. They also had to provide shims for mixins to contribute to dynamically generated methods.

I sat down and worked out exactly what was needed from a framework, and no more. The result is a module that is about 1574 lines of code (with comments.)

## Implementation

Clay introduces a method ensemble to both `oo::class` and `oo::object` called `c1ay`. This ensemble handles all of the interactions within the framework. It also limits itself to two variables inside of objects: `c1ay` and `c1aycache`.

In a bulleted list Clay:

- Stores structured data
- Permits access from an object to the structured data of it's constituent classes.
- Manages method delegation
- Enforces policies governing mixin interactions
- Enforces contracts between classes and objects to initialize scalar variables, dicts, and arrays

There is an optional language extension which adds:

- Method Ensembles
- Syntactic sugar around declaring arrays, dicts, and scalars.

The central concept is that inside of every object and class (which are actually objects too) is a dict called `c1ay`. What is stored in that dict is left to the imagination. But because this dict is accessed via a public method, we can share structured data between object, classes, and mixins.

## Structured Data

Clay provides a mechanism for structured data to be shared between an object and the classes that make that object. If you are familiar with my work with `oo::meta` and `tool`, structured data in my style of frameworks is nothing new. In previous efforts, I used a central database to store that information per-class.

There are challenges to that approach. Each class had to flesh out it's entire dictionary to search it properly. Data was constantly becoming stale as new packages were loaded. Workarounds were developed, and performance enhancing parlor tricks, but the system was brittle.

Clay has no central database. Instead, it uses a stylized set of method interactions combined with introspection that TclOO already provides. Together they allow object to perform on-the-fly searches. On-the-fly searches mean that the data is never stale, and we avoid many of the sorts of collisions that would arise when objects start mixing in other classes during operation.

The clay methods for both classes and objects have a `get` and a `set` method. For objects, `get` will search through the local `c1ay` dict. If the requested leaf is not found, or the query is for a branch, the system will then

begin to poll the `clay` methods of every class that implements the object, all of the classes that have been mixed in, as well as all of the ancestors of those classes.

## Keeping Branches and Leaves Straight

A few quick words on notation. In most respects, `clay` is like any other `dict`. Each value can, itself, be another `dict`. Occasionally you will see that intended branches on a tree end with a directory slash (`/`). And occasionally you will see intended leaves end with a colon (`:`). This is a guide for the tool that builds the `dicts` to tag what parts of a `dict` are intended to be branches and which are intended to be leaves.

Inputting one value at a time with `clay set` the system can determine what is a branch or a leaf from the arguments, and branch marking can be ignored:

```
::oo::class create ::foo { }
::foo clay set property color blue
::foo clay set property shape round

set A [::foo new]
$A clay get property
    {color blue shape round}

$A clay set property shape square
$A clay get property
    {color blue shape square}
```

But when you start storing blocks of text, guessing what field is a `dict` and what isn't can get messy. Fortunately, when in doubt the system assumes that the final value given to `clay set` is intended to be a leaf:

```
::foo clay set description {A generic thing of designated color and shape}

$A clay get description
    {A generic thing of designated color and shape}

::oo::class create ::bar {
  superclass foo
}
::bar clay set description \
  {A drinking establishment of designated color and shape and size}
set B [::bar new]
$B clay get description
    {A drinking establishment of designated color and shape and size}
```

Occasionally though, confusion can set in. Without a convention for discerning branches for leaves what should have been a value can be accidentally parsed as a dictionary, and merged with all of the other values that were never intended to be merge. Here is an example of it all going wrong:

```
::oo::class create ::foo { }
# Add description as a branch
::foo clay set description/ \
  {A generic thing of designated color and shape}
$A clay get description
  {A generic thing of designated color and shape}
# So far so good

::oo::class create ::bar {
  superclass foo
}
# Add a description as a branch
::bar clay set description/ \
  {A drinking establishment of designated color and shape and size}
set B [::bar new]
$B clay get description
  {A generic thing of designated color and shape establishment of}
```

If you are having trouble following, because we interpreted `description/` as a `dict` instead of a string, our merge tool broke the text into key/value pairs. It thought *A*, *thing*, *designated*, *and*, and *establishment* were the keys. If we didn't happen to give our blocks of text an even number of words, the operation would have failed with an error.

You can also see that we had to really put some work into screwing this up, but if you can take the word of someone who has been doing this for a few years, it happens.

You can also have the opposite occur, what should have been a `dict` inside of a `dict` (and thus merged) is interpreted as a leaf and the values replaced. So you will see throughout this paper, software libraries and my own personal code liberal use of the `/` and `:`.

## dicttool

`Clay` utilizes *dicttool* to do its recursive merging, and a side effect of *dicttool* is every known branch being marked with an extra key: `.` (a dot).

```
dicttool::dictmerge result \
  {option/ {color/ {default: green}}}
  . 1 option { . 1 color { . 1 default green}}
```

You can see the *dicttool* strips off the `/` and `:` from the keys. Most of the `clay` public API filters those dots out. But there are times (especially when doing merges) that you want to leave them in. For that `clay` provides a parallel set of methods (`dget` and `family`).

If you are interacting with one of these dictionaries in the wild, *dicttool* has a facility for cleaning up a merge annotated `dict` called *sanitize*.

## Example: Option Handling

Most readers of this paper have either used or tried to implement option handling. This is not the strongest argument to use clay, but it's a common and intuitive enough set of rules that I don't need to devote a page to explaining it. Plus option handling is trivial to implement any number of ways. And between tcllib and tklib we do implement it any number of ways!

We are going to create a standard that everything key the *option/* branch is considered the name of an option and every value is a dict describing properties of that option. We will assume each option needs the following fields:

type	A type keyword which is meaningful to our presentation layer.
default	The default value for the option.
values	For <b>select</b> fields, the values that are possible.
validate	A script to run to validate a new value

In the next column I lay out a quick and dirty implementation of option handling using just the clay method.

No information about specific options is actually inscribed in the class itself. It just knows to consult `c_lay`. With the info in `c_lay`, and just using the public API, the methods of the class can feed rules of its own creation. `validate` is just intended to be called and throw an error if it is going to balk at an option prior to the value going into an internal data structure.

Because this object could have other classes mixed in after creation, we don't hard code the defaults. We dig through Clay to find them, and thus, if a mixin decides the default shape is not square, and we have not configured the object, a request for the config item will return the mixin's default.

Likewise, if a mixin creates a new option, that new option enters the object's ecosystem on equal footing with the class' options. That mixin can also replace the `validate` method with its own!

Naturally, this implementation is not the most robust, so pardon the brevity:

```
::oo::class create ::foo {
  constructor args {
    my config set {*} $args
  }
  method validate {field value} {
    if {[my clay exists option/ $field/]} {
      error "Unknown option $field"
    }
    set info [my clay get option/ $field]
    if {[dict get $info type] eq "select"} {
      set values [dict get $info values]
      if { $value ni $values } {
        error "$value is invalid. Valid: $values"
      }
    }
    if {[dict exists $info validate]} {
      eval [dict get $info $validate]
    }
  }
  method config {method args} {
    my variable config
    switch $method {
      set {
        foreach {f v} $args {
          set f [string trim $f /-]
          my validate $f $v
          set config($f) $v
        }
      }
      get {
        set fld [string trim [lindex $args 0] /-]
        if {[info exists config($fld)]} {
          return $config($fld)
        }
      }
    }
    if {[my clay exists option/ $fld/ default]} {
      error "Unknown option $fld."
    }
    return [my clay get option/ $fld/ default]
  }
}

::foo clay set option/ color/ {
  type color default blue
}

::foo clay set option/ shape/ {
  type select default round values {round square}
}

::oo::class create ::bar {}
::bar clay set option/ price_of_beer/ {
  default {$4}
}

::bar clay set option/ shape/ {
  type select default rectangle
  values {rectangle round square}
}

set A [::foo new color green]
$A config get color
# green

$A config get shape
# round

$A config get flavor
# Unknown option flavor

$A mixin ::bar
$A config get price_of_beer
# $4

$A config get shape
# rectangle
```

## The Clay Dialect

The clay module also includes an optional language dialect to provide new keywords which are shorthand for Clay interactions. To use the dialect, you change from using `oo::class` and `oo::define` to using `clay::define`. `clay::define` understands all of the keywords from standard TclOO. It also adds its own:

Array	Declare an internal variable which is initialized as an associative array and populated with values on construction. Note the case.
class_method	Define a method of the class object itself that will be inherited by descendent classes, but not object instances of that class. Note that a class can have a class_method implementation that has the same name as a method implementation and they will not conflict.
clay	Interact with the class' clay storage
Dict	Declare a dict that should be populated with values on construction. (Note the case)
Ensemble	Declare the arguments and body for a sub-method of a method ensemble.
Variable	Declare a variable that should be populated with a default value on construction. (Note the case.)

In the next column you will see our option handling example re-implemented using Clay's notation. I think it looks better, but as you saw on the previous page, you can get the same behavior with just the `clay` method and otherwise pure TclOO.



```

proc ::clay::define::Option {name info} {
    set class [class_current]
    dict for {f v} $info {
        $class clay set option $f $v
    }
}

::clay::define ::foo {
    Option color {type color default blue}
    Option shape {
        type select default round
        values {round square}
    }
    Dict config {}

    constructor args {
        my config set {*}$args
    }

    method Validate {field value} {
        if ![my clay exists option/ $field/] {
            error "Unknown option $field"
        }
        set info [my clay get option/ $field]
        if {[dict get $info type] eq "select"} {
            set values [dict get $info values]
            if { $value ni $values} {
                error "$value is invalid. Valid: $values"
            }
        }
        if {[dict exists $info validate]} {
            eval [dict get $info $validate]
        }
    }

    Ensemble config::set {args} {
        dict for {f v} $args {
            set f [string trim $f /-:]
            my Validate $f $v
            dict set config $f $v
        }
    }

    Ensemble config::get {field} {
        my variable config
        set field [string trim $field /-:]
        if {[dict exists $config $field]} {
            return [dict get $config $field]
        }
        if ![my clay exists option $field default] {
            error "Unknown option $field."
        }
        return [my clay get option $field default]
    }
}

::clay::define ::bar {
    Option price_of_beer {default {$4}}
    Option shape {
        type select default rectangle
        values {rectangle round square}
    }
}

set A [::foo new color green]
$A config get color
# green

$A config get shape
# round

$A config get flavor
# Unknown option flavor

$A mixin ::bar
$A config get price_of_beer
# $4

$A config get shape
# rectangle

```

## Creating Your Own Dialects

Clay is built using the `oo::dialect` module from Tcllib. `oo::dialect` allows you to either add keywords directly to clay, or to create your own metaclass and keyword set using Clay as a foundation.

In the prior example I used it to actually invent a new keyword for clay called `option`.

The `oo::dialect` system creates a namespace, and any commands in that namespace can be invoked within the body of a `metaclass::define` command. A command called `class_current` keeps track of the class that was being modified.

The `oo::dialect` system also creates two classes `metaclass::class` and `metaclass:object`. Every class created with `metaclass::define` will automatically be an descendent of `metaclass::object`.

`metaclass::class` is helpful if you want to add methods to your meta class' classes. For instance, to hijack the `unknown` mechanism in TciOO to emulate Tk widgets.

The `metaclass::define` also has one feature that `oo::define` lacks. It will happily deduce that the class you are referencing does not exist yet, and create it. For large libraries I find this helps because you may have large classes that are built up over several source files.

## Dicts Vs. Flat Lists

There is an argument to be made that dicts of dicts are overkill. Many of the functions that Clay is trying to perform would be better done by hard coded values returned by methods, or streams of values in lists. And honestly, there is nothing in Clay that prevents you from doing so in your particular application.

Not to harp on option handling, but if you pick apart the megawidget class in Tk, there is a method `GetSpecs` which returns a list of lists, and each of those lists is expected to be 4 elements, and you have to squint to figure out which element is doing which.

What would you rather encounter buried deep within the `/library` file system when debugging?

This:

```
method GetSpecs {} {
  set result [next]
  lappend result {-cursor cursor Cursor {}}
  lappend result {-takefocus takeFocus \
    TakeFocus ::ttk::takefocus}
  return $result
}
```

Or this:

```
Option cursor {
  name cursor
  class Cursor
  command {}
}
Option takefocus {
  name takeFocus
  class TakeFocus
  command ::ttk::takefocus
}
```

And if you ever want to just emulate the old `GetSpecs` method:

```
method GetSpecs {} {
  set optinfo [my clay get options/]
  set result {}
  dict for {name info} $optinfo {
    lappend result [list -${name} \
      [dict get $info name] \
      [dict get $info class] \
      [dict get $info command] \
      [dict get $info validate]]
  }
  return $result
}
```

## Method Delegation

It is sometimes useful to have an external object that can be invoked as if it were a method of the object. Clay provides a `delegate` ensemble method to perform that delegation, as well as introspect which methods are delegated in that manner. All delegated methods are marked with html-like tag markings (`<>`) around them. Behind the scenes we are simply using the

`oo::objdefine` *forward* mechanism:

```
foreach {stub object} $args {
  set stub <[string trim $stub <>>
  dict set clay delegate/ $stub $object
  oo::objdefine [self] forward ${stub} $object
  oo::objdefine [self] export ${stub}
}
```

In this example, we will be using delegation to provide an abstraction to a raw database object created by sqlite:



```

::clay::define example {
  variable buffer
  constructor {filename} {
    # Build a database connection
    set obj [namespace current]::db
    sqlite3 $obj $filename
    # Delegate the counter
    my delegate <db> $obj
  }
  method users {} {
    set result {}
    my <db> eval {select distinct username from
users} {
      lappend result $username
    }
    return $result
  }
  method userid {username} {
    set stmt {select userid from users where
username=:username or userid=:username}
    if {![my <db> exists $stmt]} {
      return -1
    }
    return [my <db> onecolumn $stmt]
  }
}

set A [example new ~/data/example1.sqlite]
set B [example new ~/data/example2.sqlite]
foreach user {$A users} {
  set usermap($user) [$B userid $user]
}

```

1

Pay special attention to the `<db> eval`. The `eval` operator of `sqlite` is a very complex animal that interacts with local variables. Emulating that without using `oo::objdefine forward` is very difficult. Another feature is the fact that we build an `sqlite` object instance inside the object's namespace. When the object is destroyed, the `sqlite` instance will be cleaned up automatically for us.

## Mixin Interaction Policies

Clay introduces several policies that sort out complex interactions between mixins. It also provides shims for scripts to fire off in response to mixin events.

Developers are free to use or ignore this feature. To use the mixin system, simply use the `clay mixin ensemble` method instead of the standard `oo::objdefine mixin` mechanism.

When that method is invoked, all classes currently mixed in, about to be mixed in, or about to be removed are polled for the following values in their Clay system:

mixin/	unmap-script	Invoked if a class is about to be removed as a mixin from an object
mixin/	map-script	Invoked if a class has just been mixed into an object
mixin/	react-script	Invoked if another class has been mixed into this object while this class remains mixed in.

```

::clay::define animal {
  clay set mixin/ map-script {
    puts "[self] says [my clay get sound]"
  }
  clay set mixin/ unmap-script {
    puts "[self] no longer says [my clay get sound]"
  }
}

::clay::define cat {
  superclass animal
  clay set sound meow
}

::clay::define dog {
  superclass animal
  clay set sound woof
}

::clay::object create felix
felix clay mixin cat
                                ::felix says meow

felix clay mixin dog
                                ::felix no longer says meow
                                ::felix says woof

```

## Beyond Options for Object Configuration

During the development of `Toadhttpd` (which extends the `Tcllib httpd` module into a full general purpose webserver) I had an interesting problem with configuration. `Toadhttpd` makes use of plugins to extend the core `httpd` server. But not every plugin is used in every server, and plugins often need settings above and beyond what the core server is aware of.

In a world where we use the likes of `Tk` options to configure an object, we are more or less stuck thinking about the world as a key/value list. If a plugin needed a new option, we could just invent a new option that doesn't conflict with an existing option. (Say *dbfile*). If we have two plugins that each want an option named *dbfile*, we could make



plugins prepend the name of the plugin to the option. (Say *dispatch\_dbfile*).

```
# configure script for example.com
my configure \
  -port 80 \
  -logdir /var/log/www \
  -dispatch_class httpd::plugin.dispatch_sqlite \
  -dispatch_cache /var/cache/www \
  -dispatch_dbfile /var/cache/www/cache.sqlite \
  -security_class httpd::plugin.blackhole \
  -security_block_null_agent 1 \
  -security_dbfile /var/cache/www/blackhole.sqlite
```

For a finite number of plugins, where the developer knows ahead of time which plugins are going to need what data, it's not too bad. But if anyone has tried to configure an internet service in the last 20 years or so, *Katy bar the door*.

With clay, you can play the game in a different way. Each plugin can have its own branch in the clay data structure to play in:

```
# configure script for example.com
my clay set {
  server/ {
    port 80
    logdir /var/log/www
  }
  plugin/ {
    dispatch {
      class httpd::plugin.dispatch_sqlite
      cache /var/cache/www
      dbfile /var/cache/www/cache.sqlite
    }
    security {
      class httpd::plugin.blackhole
      block_null_agent 1
      dbfile /var/cache/www/blackhole.sqlite
    }
  }
}
```

Rather than a random jumble of fields and values, you get a sense of not only what the setting is, but where it will be used. This example is a bit too simple, but if you read my paper on *The Httpd Module and Toadhttpd* you will see the power of passing structured data during object evolution put into action.

## Structured Data Inheritance

In a complex system where a rule has to be written to handle a range of different classes of objects, it is often useful to be able to refer to some meta-information within the object. It is also helpful to have that meta-information inherited along with the methods.

---

## Class Inheritance.

The simplest kind of inheritance is the type that we naturally think should happen in a class structure, where each class has a number of ancestors.

In this example we are creating the taxonomic classification of the common housecat. In that classification, we are seeding some useful traits that will be passed along to descendants of the class above.

```
::clay::define animal {
  clay set tkingdom Animalia
  clay set has_spine 0
}
::clay::define vertebrate {
  superclass animal
  clay set torder Chordata
  clay set has_spine 1
}
::clay::define mammal {
  superclass vertebrate
  clay set tclass Mammalia
  clay set has_fur 1
}
::clay::define carnivore {
  superclass mammal
  clay set torder Carnivora
}
::clay::define feline {
  superclass carnivore
  clay set tfamily Felidae
}
::clay::define felis {
  superclass feline
  clay set tgenus Felis
}
```

The idea being that by the time we get down to putting together the final leaf classes, the code is simply:

```
::clay::define housecat {
  superclass felis
  clay set tspecies domesticus
}
```

And should a question arise, all objects of that class can answer based in information inherited by ancestral classes.

```
housecat create Thomas
Thomas clay get torder
Thomas clay get has_fur
Thomas clay get has_backbone
```

Chordata  
1  
1

---

## Mixin Inheritance

With Mixins we have a different dimension to consider. Objects can have more than one class at a time. In the prior example, with living things, we don't need to worry about



intrinsic properties of the object changing during the creature's lifetime. Or at the very least, we tend to focus on attributes that really shouldn't change. If I wanted to take species and divide down further to breeds of cat, it's just one more layer of descendent below species.

```
::clay::define siamese {
  superclass housecat
  clay set coat_length short
  clay set ear_shape pointy
}
```

But outside of the realm of living things, we tend to have objects that are a collection of parts. Those parts can change. And those parts can radically alter the behavior of system as a whole.

Before I go off on a tangent about swapping mixins and whatnot at create time, Clay treats mixins as just another class that the object inherits from. Mixins are included in `clay ancestors` according to the following rule:

```
set clayorder [::clay::ancestors \
  [info object class [self]] \
  {*}[info object mixins [self]]]
```

Where `clay::ancestors` is defined as:

```
proc ::clay::ancestors args {
  set result {}
  set queue {}
  foreach class [lreverse $args] {
    lappend queue $class
  }

  # Rig things such that that the top superclasses
  # are evaluated first
  while {[length $queue]} {
    set tqueue $queue
    set queue {}
    foreach qclass $tqueue {
      foreach aclass \
        [::info class superclasses $qclass] {
        if { $aclass in $result } continue
        if { $aclass in $queue } continue
        lappend queue $aclass
      }
    }
    foreach item $tqueue {
      if { $item ni $result } {
        lappend result $item
      }
    }
  }
  return $result
}
```

Essentially, the most recent mixin (the one as the last argument) gets first crack, and then its ancestors. We then move on to the next most recent mixin, and repeat. And when we run out of mixins, we then evaluate the class of the object itself, and then look through its ancestors.

The long and short of the algorithm is that properties will take hold at the same spot a method would in TclOO's inheritance mechanism.

## Mixins as Configuration Options

One of my side projects is developing a text based role playing engine. If you are familiar with Dungeons & Dragons (or whatever the kids are calling it these days...) each Player and Non-Player Character (NPC) have several "slots" that affect how they interact with the world:

Race	Species of the player. There are special rules for each race when building the character as well as special abilities conferred by race. Options: human, elf, dwarf, halfling
Class	What is vocation of the adventurer. These affect weapons and spells they are allowed to use, as well as how the character conducts himself in combat. Options: warrior, wizard, cleric, thief, bard
Alignment (Law)	The default manner in which the player's character is expected to interact with the world. Choices are Options: Lawful, Neutral, Chaotic
Alignment (Moral)	The default manner in which the player's character is expected to interact with the world. Choices are Options: Good, Neutral, Evil

For now we'll ignore the further matrixes we would need to develop for weapons, armor, magical items, magical effect, and whatever the Game's master has decided to inflict the play with because she is annoyed with them.

The way that I've found to solve this problem is by artificially imposing "slots" on mixins. Mixins that are mutually exclusive are considered options on one of those slots. For really exotic scenarios where we hybridize more than one option, we can

actually mix them in together on the same slot.

The order in which mixins take hold is determined by the order in which the slots were added to a dictionary. It's not a perfect system, but it's a system.

Our class structure for the object itself becomes deceptively simple: there is only once class of object. That class has only the basic methods to load mixins and perform low level interactions with the framework.

To create an object, I just spawn off the undifferentiated object, and then mix the heck out of it:

```
::stage::object new {
  uuid 7d7c0261-5a7a-4fd9-946e-c23d59d70b70
  name {The Player}
  mixin {
    core ::stage::avatar
    race ::stage::race.human
    class ::stage::class.cleric
    delegate {
      db ::db
      stage ::GAME
    }
    alignment {
      ::stage::alignment.lawful
      ::stage::alignment.good
    }
  }
}
```

Internally, that object's constructor just looks for certain keywords to tell it important behavioral bits, and just writes everything else to the clay data structure:

```
clay::define ::stage::object {
  constructor {claydat} {
    if {![dict exists $claydat uuid]} {
      dict set claydat uuid \
        [::uuid::uuid generate]
    }
    dict for {f v} $claydat {
      if {$f in {delegate mixin}} continue
      my clay set $f $v
    }
    my clay delegate \
      {*}[dict getnull $claydat delegate]
    my clay mixinmap \
      {*}[dict getnull $claydat mixin]
    my generate {msg_subject object_created}
  }
}
```

If you think this example is a little contrived, here is a class method to determine which mixin to slot in based on the local environment from Practcl:

```
oo::objdefine ::practcl::toolset {
  method select object {
    # Select the toolset to use for this project
    if {[object define exists toolset]} {
      return [object define get toolset]
    }
    set class [object define get toolset]
    if {$class ne {}} {
      # The object configuration state a
      # class to use
      $object mixin toolset $class
    } else {
      # Ok... we don't have anything pencilled
      # in guess base on he environment
      if {
[info exists ::env(VisualStudioVersion)]
      } {
        $object clay mixinmap \
          toolset ::practcl::toolset.msvc
      } else {
        $object clay mixinmap \
          toolset ::practcl::toolset.gcc
      }
    }
  }
}
```

For Httpd we use mixins to inject behaviors into httpd::reply instances:

```
method dispatch {newsock datastate} {
  my variable chan request
  try {
    set chan $newsock
    chan event $chan readable {}
    chan configure $chan \
      -translation {auto crlf} -buffering line
    my clay mixinmap \
      {*}[dict getnull $datastate mixin]
    my clay delegate \
      {*}[dict get $datastate delegate]
    my reset
    set request [my clay get dict/ request]
    foreach {f v} $datastate {
      if {$f in {mixin delegate}} continue
      if {[string index $f end] eq "/" } {
        my clay merge $f $v
      } else {
        my clay set $f $v
      }
      if {$f eq "http"} {
        foreach {ff vf} $v {
          dict set request $ff $vf
        }
      }
    }
  }
  my Session_Load
  my Log_Dispatched
  my Dispatch
} on error {err errdat} {
  my error 500 $err \
    [dict get $errdat -errorinfo]
  my DoOutput
}
```

And note that in the case of Practcl and Httpd, those methods are not called by the constructor. The power of TcIOO is that mixins can happen at any time in the lifecycle of an object.



One of the nifty things that flexibility allows is to make objects serve as both HTTP and SCGI content sources.

At the tail end of the `httpd::server.scgi` class' Connect method is the following snippet:

```
set pageobj [::httpd::reply create \  
  ::httpd::object::$uuid [self]]  
dict set reply mixin \  
  protocol ::httpd::protocol.scgi  
$pageobj dispatch $sock $reply
```

Essentially with one mixin we can alter the the headers returned by `::httpd::reply` such that it's compatible with an SCGI proxy.

There are a few things to pay attention to in the three examples I've cited. The `stage::object` constructor knows it is always getting the first crack at the clay data structure. However, in the `httpd::reply` dispatch method, we are already part of an established object. You will see that the

mechanism for merging clay data is a tad more sophisticated so that leaves are replaced and branches are merged. We also have special handling for one branch (*http*) which is copied over to the object's request dictionary.

The point I am trying to make is that all of these projects use clay while still being true to the domain specific rules of the problem they are trying to tackle.

## Conclusion

I goal in this paper was to introduce Clay, as well as the problems it tries to solve. By leveraging conventions already familiar to Tclers, I hope you find it easy to pick up and comfortable to code with. And even if you don't, it can be doing plenty of cool things behind the scenes without causing you a lot of fuss an bother.

## For More Information

Clay is distributed as part of Tcllib:  
<https://core.tcl-lang.org/tcllib>

The manual page can be directly access from the web:

<https://core.tcl-lang.org/tcllib/doc/trunk/embedded/www/tcllib/files/modules/clay/clay.html>

There is also a development version that is being adapted into text adventure game engine and natural language parsing system:

<http://fossil.etoyoc.com/fossil/clay>  
<http://chiselapp.com/user/hypnotoad/repository/clay>

## Cited Works

Cover and clip art:  
Celtic Stencil Designs CD-ROM and Book  
Co Spinhoven  
<http://store.doverpublications.com/0486996786.html>

The Httpd Module and Tclhttpd  
Sean Woods  
[http://www.etoyoc.com/yoda/papers/tcl2018.Httpd\\_Paper.pdf](http://www.etoyoc.com/yoda/papers/tcl2018.Httpd_Paper.pdf)