Adventures in TclOO

Donal Fellows,
University of Manchester

Abstract

I have been working on a number of small projects that use TclOO in fairly complex ways, and this paper will cover some of them. In particular, it will look at its use for building more advanced APIs in the areas of interfacing to the HTTP protocol for use with REST services, for mapping relational databases into Tcl, and for supporting metadata on TclOO objects themselves.

This paper is a collection of write-ups for some small projects in Tcl that I have been working on in the past year that all leverage the TclOO object system[1]. The first is a wrapper for the standard http package that makes it easier to write clients for particular types of web services, the second is a way of representing the contents of a relational database in Tcl and interacting with it, and the third is a mechanism for adding extra information to classes and their contents.

Note that the first package described here is actually implemented so as to be usable in Tcl 8.5, whereas the other two both require Tcl 8.6.

Simple REST Interface

Representational State Transfer[2] (or REST) is a way of providing services over the web in a way that works well with the fundamental architecture of the web. In particular, it focuses on the use of standard HTTP verbs[3] (including whether or not they are supposed to be idempotent) and resources characterized by URLs that can be retrieved in multiple formats (with content negotiation to decide which format to use). For example, a collection of pizzas might be represented by the URL `http://example.org/pizzas`, that you would do a normal GET of to get a list of URLs as some format like HTML, XML, JSON, YAML, Tcl list, S-expression, etc. A URL such as `http://example.org/pizzas/pepperoni` would represent each pizza on the list, and you would be able to GET that URL to get a detailed description of the pizza and remove the pizza from the list by doing a DELETE on the URL. A pizza of known name would be created or altered by doing a PUT on its URL, or a POST could be done on the collection of pizzas to create a new one (of the chef's choice); the response would be an HTTP redirect to the newly created pizza.

There are many ways to interact naturally with a REST web service – it is even feasible to just treat one as a collection of plain web pages – but the standard Tcl http package does not make this particularly easy. Key things like handling of HTTP verbs, redirects and content negotiation are concealed behind an interface that both conceals critical features and reveals much of its implementation. That's where my REST support code comes in.

Code and Discussion

It simply consists of a TclOO class that provides methods that implement each of the common HTTP verbs required for RESTful service interaction (GET, PUT, POST and DELETE). These methods in turn delegate their behaviour to a worker method that takes care of the nitty gritty details of things like redirections. It also makes it much easier to specify an alternate preferred set of content types for a particular request (e.g., so you could get a directory either as a listing of its contents or as a zipped archive).

For the GET method, since it is common to not provide an entity with this method, it does a join on the arguments provided with "/" as a separator, adding these on to the base URL set

in the constructor. This is then passed straight on to the fundamental DoRequest method, which is a bounded loop that performs basic requests using the standard http library until success or a recoverable failure is reached. The actual decision of what to do about a particular redirect is taken by another method, OnRedirect.

```
method DoRequest {method url {type ""} {value ""}} {
  for {set reqs 0} {$reqs < 5} {incr reqs} {
    if {[info exists tok]} {
      http::cleanup $tok
    }
    set tok [http::geturl $url -method $method \
        -type $type -query $value]
    if {[http::ncode $tok] > 399} {
      set msg [my ExtractError $tok]
      http::cleanup $tok
      return -code error $msg
    } elseif {[http::ncode $tok] > 299
        || [http::ncode $tok] == 201} {
      try {
        set location [dict get [http::meta $tok] Location]
      } on error {} {
        http::cleanup $tok
        error "missing a location header!"
      }
      my OnRedirect $tok $location
    } else {
      set s [http::data $tok]
      http::cleanup $tok
      return $s
    }
  }
  error "too many redirections!"
}

method OnRedirect {tok location} {
  upvar 1 url url
  set url $location
  # By default, GET doesn't follow redirects; the next
  # line would change that...
  ###return -code continue
  set where $location
  set len [string length "$base/"]
  if {[string equal -length $len $location "$base/"]} {
    set where [string range $where $len end]
    return -level 2 [split $where "/"]
  } else {
    return -level 2 $where
  }
}
```

```
method ExtractError {tok} {
  return [http::code $tok],[http::data $tok]
}
```

With the aid of the above methods, the definition of the general method for handling GET requests is just this:

```
method GET args {
  return [my DoRequest GET $base/[join $args "/"]]
}
```

The equivalent for other HTTP verbs is very similar, though with additional arguments defined due to the need to control how values are uploaded.

Usage

In subclasses of this general REST support class, I can just do a read/write accessor method like this:

```
method status {{status ""}} {
  if {$status eq ""} {
    return [my GET status]
  }
  my PUT status text/plain $status
  return
}
```

You don't get much easier than that without adding in code to directly work with a WADL file published by the service, and that's not so commonly published for REST services.

This was used to rapidly prototype an interface for a service I was developing. That service had a very large service API (42 methods, most of which have renderings as both SOAP and REST styles simultaneously) so the use of a testing tool was vital. While I already had an existing (non-Tcl) infrastructure for checking the SOAP interface[1], I needed something I could work with for the REST interface. In particular, I needed to be able to build things up piece-by-piece so I could check what I was doing as I was doing it. By turning a reasonably complex sequence of operations with the http package into a simple method call, it made

[1] I also uncovered a few problems with the Tcl-WS[4] packages, subsequently fixed of course, when trying to do this in Tcl.

it much easier for me to focus on the debugging of my service and it also allowed me to very rapidly throw up a GUI (thanks, Tk!) for the purposes of demonstration.

Object Relational Mapping

When a database is used with Tcl, it is most common to do this by simply issuing SQL queries and statements to the database using one of the many existing interfaces. Indeed, TDBC[5] is a standardization of these interfaces to promote best practices in database access so that it can become easier to write cross-platform code that works with many databases. However, using such interfaces still requires the script author to understand SQL[6]. One way to relax this requirement is to map a database as a collection of classes and objects; the classes correspond approximately to the tables in the database schema and the objects to the rows in those tables. This is a common approach in many languages[7][8][9], especially where there are strict static types, and often involves a complex compilation step that generates the classes and queries from the database. But I wanted to see how much I could do in an entirely dynamic way while leveraging TclOO and TDBC to do the difficult parts for me. It turns out you can do a lot!

I opted in this work to make objects that reflected the database rather than the reverse, this being driven largely by the fact that databases tend to have more detailed information about relationships between tables and columns (plus the types of the columns) than is actually required for Tcl. After all, Tcl is quite happy with almost anything as a value. Because of this, my aim is to have a package of TclOO classes such that it could be told to attach to a TDBC database connection and have a class created for each (non-metadata) table. Constructors would correspond to the use of SQL queries or inserts, and individual columns would map to methods that read and write the values inside the corresponding columns. There is an explicit one-to-one mapping between table names and class names, and between the names of columns and methods; the unexposed support methods have names that it is wholly unreasonable to use as column names.

Code and Discussion

The ORM package is composed of four classes that represent the overall database, a table in that database, and two that represent different types of row.

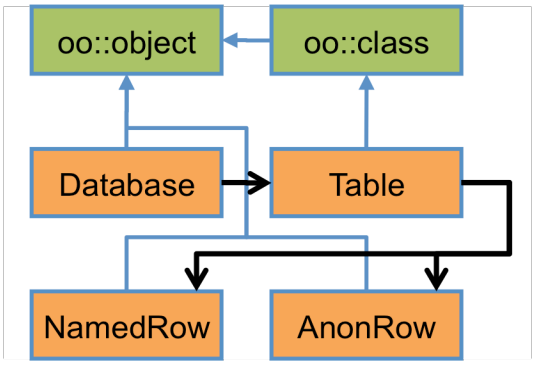

Figure 1: ORM class diagram

Figure 1 shows how the classes related to each other (the standard TclOO classes are included in green for clarity). The blue links indicate inheritance, and the black links indicate where one class is responsible for creating instances of another.

The Database class encapsulates the whole database, or rather the TDBC connection, and it acts as a collection of tables. It also generates the particular table classes with the help of introspection on the database.

```
oo::class create ORM::Database {
  variable db tableClasses dying
  self {
    variable classes
    method ClassFor {category default {class ""}} {
      if {$class ne ""} {
        set classes($category) $class
      } elseif {![info exists classes($category)]} {
        return $default
      }
      return $classes($category)
    }
    forward classOfTables \
        my ClassFor table ::ORM::Table
    forward classOfNamedRows \
        my ClassFor namedRow ::ORM::NamedRow
    forward classOfAnonRows \
        my ClassFor anonRow ::ORM::AnonRow
  }
  constructor {databaseHandle} {
    set db $databaseHandle
    oo::objdefine [self] export GetRowClass
```

```
    $db transaction {
      dict for {table ?} [$db tables] {
        dict set tableClasses $table \
            [my MakeAMappedTable $table]
      }
    }
    oo::objdefine [self] unexport GetRowClass
  }
  destructor {
    set dying "dying"
    foreach tbl $tableClasses {$tbl destroy}
  }
  method transaction script {
    $db transaction {uplevel 1 $script}
  }
  method tables {} {
    dict keys $tableClasses
  }
  method table {table args} {
    if {![llength $args]} {
      return [dict get $tableClasses $table]
    }
    tailcall [dict get $tableClasses $table] {*}$args
  }
  method MakeAMappedTable {table} {
    [[self class] classOfTables] create $table \
        [self] $db $table
  }
  method GetRowClass {type} {
    [self class] classOf${type}Rows
  }
}
```

The Table class is the core of the ORM package as it contains almost all of the complexity. In particular, the constructor uses introspection on the underlying database table to discover not just the collection of columns, but also their natures. Unlike most object-relational mapping technologies, we do not need to do much to handle the types of the columns (though the current prototype does ignore SQL NULLs) but the nature of columns is still important because it is important to use primary key and foreign key information to manage the mapping between tables; that enables the transparent following of links between tables, coupling row objects together smoothly. (This sample code omits much to keep things short.)

```
oo::class create ORM::Table {
  superclass oo::class
  variable db dbHandle table sql id2obj idColumn \
      columns foreignKeyMap
  constructor {mappedDB connection tableName} {
    set db $mappedDB
    set dbHandle $connection
    set table $tableName
    set pkinfo [$connection primarykeys $table]
    set fkinfo [$connection foreignkeys -foreign $table]
    if {[llength $pkinfo] == 1} {
      set idColumn \
          [dict get [lindex $pkinfo 0] columnName]
      oo::define [self] superclass \
          [$db GetRowClass Named]
    } else {
      set idColumn ""
      oo::define [self] superclass \
          [$db GetRowClass Anon]
    }
    array set id2obj {}
    dict for {c info} [$connection columns $table] {
      if {$c eq $idColumn} {
        set name "ORM.Access.ID"
        set target {}
      } else {
        lappend columns $c
        set target [my GetFKTarget $fkinfo $c]
        if {[llength $target]} {
          set name "ORM.Access.Linked"
          lassign $target targetTable targetKey
          dict set foreignKeyMap $c $target
        } else {
          set name "ORM.Access.Simple"
          set sql(update,$c) \
              [my MakeUpdateOfRowColumn $c]
        }
      }
      oo::define [self] forward $c my $name $c {*}$target
      oo::define [self] export $c
    }
    set sql(query,all) [my MakeQueryForAllRows]
    if {$idColumn ne ""} {
      set sql(query,byID) \
          [my MakeQueryForRowByIdentifier]
      # etc...
    }
  }
  unexport create new
  method GetFKTarget {descriptor sourceColumn} {
    foreach fkDesc $descriptor {
      dict with fk {
        if {$foreignColumn eq $sourceColumn} {
          return [list $primaryTable $primaryColumn]
        }
      }
    }
  }
  method MakeQueryForRowByIdentifier {} {
    format {SELECT * FROM "%s" WHERE "%s" = :id} \
        $table $idColumn
```

```
  }
  # etc...
  method findById {id} {
    if {![info exist id2obj($id)]} {
      $dbHandle foreach -as dicts $sql(query,byID) row {
        set id2obj($id) [my MakeRowForId $row $id]
        break
      }
    }
    return $id2obj($id)
  }
  method foreach {varName script} {
    upvar 1 $varName v
    $dbHandle foreach -as dicts $sql(query,all) row {
      set id [dict get $row $idColumn]
      if {![info exists id2obj($id)]} {
        set id2obj($id) [my MakeRowForId $row $id]
      }
      set v $id2obj($id)
      uplevel 1 $script
    }
  }
  method MakeRowForId {rowDictionary identity} {
    tailcall my new [namespace which my] \
        $rowDictionary $identity
  }
  method MakeRowWithoutId {rowDictionary} {
    tailcall my new [namespace which my] $rowDictionary
  }
  method mappedDB {args} {
    if {![llength $args]} {return $db}
    tailcall $db {*}$args
  }
}
```

Database rows are represented by subclasses of the two row classes. One is for rows where we have a mapped identity, and the other is for rows where that was not possible (e.g., because the primary key consists of multiple columns). I show just one of these classes here; the other is a stripped-down version of it.

```
oo::class create ORM::NamedRow {
  variable contents tbl id
  constructor {table row identity} {
    array set contents $row
    set tbl $table
    set id $identity
  }
  destructor {
    $tbl RemoveRow $id
  }
  method ORM.writeback {column value} {
    tailcall $tbl SetRowColumn $id $column $value
  }
  method ORM.Access.Simple {c args} {
    if {[llength $args]} {
      lassign $args value
      set contents($c) [my ORM.writeback $c $value]
    }
    return $contents($c)
  }
  method ORM.Access.Linked {c targTbl targCol args} {
    if {[llength $args]} {
      lassign $args value
      # Error checking elided for clarity
      set cls [$tbl mappedDB table $targTbl]
      set otherId [set [info object namespace $value]::id]
      set contents($c) [my ORM.writeback $c $otherId]
      return [$cls findById $contents($c)]
    }
    $tbl mappedDB table $targTbl \
        findById $contents($c)
  }
  method ORM.Access.ID {c} {
    return $contents($c)
  }
}
```

Usage

To show how this mapping might be used, let us consider a simple order database. Each order has its ID, of course, and a description of the order, and it also has references to the customer who is paying for the order and where it is to be dispatched to. They are linked as in Figure 2.

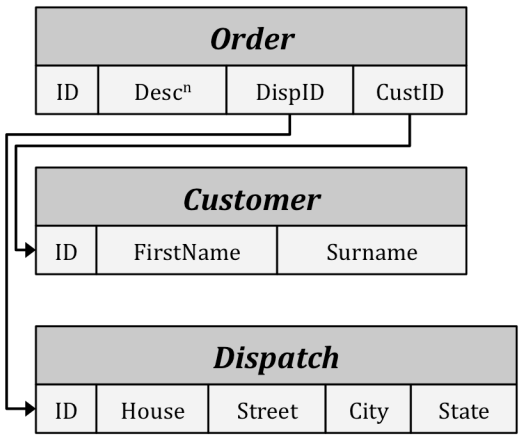

Figure 2: Order dispatch database schema

Specifically, this table definition is used for this example:

```
CREATE TABLE customers(
  id            INTEGER PRIMARY KEY,
  firstname     TEXT,
```

```
    surname         TEXT)
CREATE TABLE dispatch(
    id              INTEGER PRIMARY KEY,
    house           INTEGER,
    street          TEXT,
    city            TEXT,
    state           TEXT)
CREATE TABLE "order"(
    id              INTEGER PRIMARY KEY,
    customer        INTEGER NOT NULL,
    dispatch        INTEGER NOT NULL,
    description     TEXT,
    CONSTRAINT fk_customer
        FOREIGN KEY (customer)
        REFERENCES customers(id)
        ON DELETE CASCADE,
    CONSTRAINT fk_dispatch
        FOREIGN KEY (dispatch)
        REFERENCES dispatch(id)
        ON DELETE CASCADE)
```

To print out a listing of the contents of this (admittedly very simple) database with ORM, you would just need to write code like this:

```
set conn [tdbc::sqlite3::connection new "mydb.sqlite3"]
ORM::Database create db $conn
db table order foreach ordr {
    puts "Order #[$ordr id]"
    puts "Customer: [[$ordr customer] firstname]\
        [[$ordr customer] surname]"
    puts "Address: [[$ordr dispatch] house]\
        [[$ordr dispatch] street]"
    puts "Address: [[$ordr dispatch] city],\
        [[$ordr dispatch] state]"
    puts "Description:\n\t[$ordr description]"
    puts ""
}
```

The fact that the order class's customer and dispatch methods both return objects on queries is deduced automatically from inspection of the foreign key constraints, together with the way that they map to primary key columns in the other tables.

Annotations

The third of these short "adventures" is into adding metadata to TclOO classes through annotations. The aim of this is to allow all declarations in a class, together with the overall class itself, to have extra information added to them that was not originally envisaged as part of the TclOO specification. The key goal of this is to allow the definition of new annotations simply by creating an appropriate subclass (the instances of the class being created automatically during the annotation process).

The inspiration for this is the annotation mechanisms present in Java[10], C#[11] and Python[12], where they serve many purposes. However, I have not focussed on adding annotations to overall individual objects so far because of the wider variety of ways in which objects are created in practice, relative to classes.

Code and Discussion

The core of the annotation system is an array in the package's namespace that maps from class names to the collection of annotations on things related to that class. That collection is implemented as a dictionary (keyed by the name of the annotation) to a list of annotation objects with that name on that class.

A lookup command is provided to search the collection of annotations; this command is integrated into Tcl's info command:

```
proc oo::InfoClass::annotations {
    class {annotation ""} args} {
  upvar #0 ::oo::Annotations::classInfo info
  set class [uplevel 1 [list namespace which $class]]
  if {$annotation eq ""} {
    if {![info exists info($class)]} return
    return [dict keys $info($class)]
  } elseif {
    ![info exists info($class)]
    || ![dict exists $info($class) $annotation]
  } then {
    return -code error \
        "unknown annotation \"$annotation\""
  }
  set result {}
  foreach h [dict get $info($class) $annotation] {
    try {
      $obj describe result {*}$args
    } on error msg {
      return -code error $msg
    }
  }
  return $result
}
```

Insertion of an annotation into the array is done by the combination of a unknown-command handler that builds the annotation when necessary, and rewritten versions of the definition commands that add in the current accumulated

annotation set to the array once it is determined what the annotations actually apply to.

The unknown handler is this:

```
proc DefineUnknown {cmd args} {
  if {[string match @* $cmd]} {
    try {
      variable subject [lindex [info level -1] 1]
      variable currentAnnotators
      lappend currentAnnotators \
          [Annotation.[string range $cmd 1 end] new \
          "class" {*}$args]
      return
    } on error msg {
      return -code error $msg
    }
  }
  # Use some knowledge of how TclOO really works...
  tailcall ::oo::UnknownDefinition $cmd {*}$args
}
namespace eval ::oo::define [list namespace unknown \
    [namespace which DefineUnknown]]
```

The definition commands are spliced like this:

```
namespace eval RealDefines {}
apply [list {} {
  foreach cmd [info commands ::oo::define::*] {
    set tail [namespace tail $cmd]
    set target ::oo::Annotations::RealDefines::$tail
    rename $cmd $target
    proc $cmd args "
      ::oo::Annotations::ClassDefinition $tail {*}\$args
      tailcall [list $target] {*}\$args
    "
  }
} [namespace current]]
```

This is supported by the ClassDefinition procedure:

```
proc ClassDefinition {operation args} {
  variable currentAnnotators
  if {![info exists currentAnnotators]} return
  variable subject
  variable classInfo
  try {
    foreach a $currentAnnotators {
      set name [$a name]
      $a register $operation {*}$args
      if {
        ![info exists classInfo($subject)]
        || ![dict exists $classInfo($subject) $name]
      } then {
        dict set classInfo($subject) $name {}
      }
      dict lappend classInfo($subject) $name $a
      set currentAnnotators \
          [lrange $currentAnnotators 1 end]
    }
  } on error msg {
    foreach a $currentAnnotators {$a destroy}
    return -level 2 $msg
  } finally {
    unset currentAnnotators
  }
}
```

The final part of the annotation package is the base annotation classes themselves.

```
::oo::class create annotation {
  unexport create
  variable annotation Type Operation
  constructor {type args} {
    set Type $type
    my MayApplyToType $type
    my RememberAnnotationArguments $args
  }
  method MayApplyToType type {
    throw ANNOTATION \
        "may not apply this annotation to that type"
  }
  method MayApplyToOperation operation {
    throw ANNOTATION \
        "may not apply that annotation to this operation"
  }
  method RememberAnnotationArguments values {
    set annotation $values
  }
  method QualifyAnnotation args {
    # Do nothing by default
  }
  method name {} {
    set name [namespace tail [info object class [self]]]
    return [regsub {^Annotation.} $name @]
  }
  method register {operation args} {
    set Operation $operation
    my MayApplyToOperation $operation
    my QualifyAnnotation {*}$args
  }
  method describe {varName} {
    upvar 1 $varName v
    lappend v $annotation
  }
}
::oo::class create classannotation {
  superclass annotation
  method MayApplyToType type {
    if {$type ne "class"} {next $type}
```

```
  }
}
```

A major class of annotations are those that are used to provide simple descriptions of parts of a class definition. To support this basic use, a class of annotations that are such descriptions is also given:

```
oo::class create Annotation.Describe {
  superclass oo::Annotations::classannotation
  variable annotation Operation method
  method MayApplyToOperation operation {
    if {$operation ni "method forward constructor"} {
      next $operation
    }
  }
  method QualifyAnnotation {name args} {
    if {$Operation eq "constructor"} {
      set method "<<constructor>>"
    } else {
      set method $name
    }
  }
  method describe {varName {name ""}} {
    upvar 1 $varName result
    if {[llength [info level 0]] == 3} {
      dict set result $method [join $annotation]
    } elseif {$method eq $name} {
      set result [join $annotation]
      return -code break
    }
  }
}
```

Usage

To use the annotations now, all you would need to do is put them on the definition, like this:

```
oo::class create foo {
  @Describe This method simply prints its arguments
  method bar args {puts $args}
}
```

After that, the annotation would be read by just using the introspection mechanism:

```
puts annotations:\t[info class annotation foo]
puts describe:\t[info class annotation foo @Describe bar]
```

Which would print out this:

```
annotations:    @Describe
describe:       This method simply prints its arguments
```

The second could have been written without the final “bar” argument, in which case it would have returned a dictionary with one entry for each method it applied to.

Just subclassing the Describe annotation class creates specialist types of description. For example, to create one just for describing side effects you might do this:

```
oo::class create Annotation.SideEffects {
  superclass Annotation.Describe
}
```

We can do the same for results, except that this time we do not want them to apply to constructors (where the result is ignored if it isn’t an error):

```
oo::class create Annotation.Result {
  superclass Annotation.Describe
  method MayApplyToOperation operation {
    if {$operation eq "constructor"} {
      throw ANNOTATE "not on a constructor"
    }
    next $operation
  }
}
```

Annotations can also be extended so they can refer to individual arguments to a method:

```
oo::class create Annotation.Argument {
  superclass Annotation.Describe
  variable annotation method argument
  constructor {type argName args} {
    set argument $argName
    next $type {*}$args
  }
  method describe {varName {name ""} {argname ""}} {
    upvar 1 $varName result
    if {[llength [info level 0]] == 3} {
      lappend result $method
    } elseif {$method eq $name} {
      if {[llength [info level 0]] == 4} {
        lappend result $argument
      } elseif {$argument eq $argname} {
        set result [join $annotation]
        return -code break
      }
    }
  }
}
```

These are applied in a completely analogous way:

```
oo::class create foo2 {
  @Describe         This has many annotations attached
```

```
    @Argument x         Ignored.
    @Argument args      To allow any number of arguments
    @Result             None.
    @SideEffects        Prints to stdout.
    method bar {x args} { puts foo }
}
```

If we introspect simply, we get a list of all the annotations that are present:

```
puts [info class annotation foo2]
```

This will print:

```
@Describe @Argument @Result @SideEffects
```

Future Work

I would like to be able to work further on the REST package so that it is better able to handle supporting the wide range of RESTful APIs found out there. In particular, the current http package creaks at the seams rather when pressed into use for this as it makes a number of assumptions that do not hold when dealing with APIs for computer consumption rather than for human consumption.

For the ORM package, I think it needs some more work so that it becomes even more natural to use before it can be considered suitable for use. In particular, there are problems with whether a row should be deleted when its corresponding object is destroyed, and also with just how eagerly objects should be created to represent database rows. After all, the aim is to make working with an *existing* database as natural as working with Tcl.

The annotations work is interesting and may lead to new things being added to TclOO in the future, though it is (as of the time of writing) significantly incomplete in that it is very awkward to create class-level annotations. This is a consequence of the fact that interceptors for the annotations are all hooked off the TclOO definition subsystem. It's also arguably the case that the axes on which annotations are currently looked up are in the wrong order (currently the order is class, annotation name, method name, etc). Some work is clearly needed in this area.

The interesting thing that can be learned from other languages though is that using annotations can lead to an interesting way for a system of objects to interact with a framework such as a web front end or GUI. This is an area where I feel I have only scratched the surface.

References

[1] Fellows, D. et al, *TIP #257: Object Orientation for Tcl*, in Tcl Improvement Proposal series, Tcl Core Team, 2005–2008.

[2] Fielding, R., Taylor, R., *Architectural styles and the design of network-based software architectures*, University of California, Irvine, CA, 2000.

[3] Fielding, R. et al, *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, in IETF Requests for Comments series, The Internet Society, 1999.

[4] Lester, G. et al, *Web Services for Tcl* package, hosted on Google Code at `http://code.google.com/p/tclws/`, 2006–2010.

[5] Kenny, K. et al, *TIP #308: Tcl Database Connectivity (TDBC)*, in Tcl Improvement Proposal series, Tcl Core Team, 2007–2008.

[6] Date, C., *A guide to the SQL standard*, Addison-Wesley, Boston, MA, 1986.

[7] Bauer, C., King, G., *Java Persistence with Hibernate*, Manning, Greenwich, CT, 2006.

[8] Roos, R., *Java Data Objects*, in Computing Reviews, vol. 45, no. 4, pp. 202, 2004

[9] Bächle, M., Kirchberg, P., *Ruby on Rails*, in IEEE Software, vol. 24, pp. 105–108, IEEE Computer Society, Los Alamitos, CA, 2007.

[10] Buckley, A. et al, *JSR 175: A Metadata Facility for the Java™ Programming Language*, in Java Specification Requests, 2002–2004.

[11] Gunnerson, E., *A Programmer's Introduction to C#, Second Edition*, Springer, New York, NY, 2001.

[12] Smith, K. et al, *PEP 318: Decorators for Functions and Methods*, in Python Enhancement Proposals series, Python Software Foundation, 2003–2004.