WyattERP: A Non-Sissy ERP Application Development Platform

Authors: Kyle Bateman Bret Barney

Development History

Original flat-file, command-line C programs

My partner and I started Action Target in 1986. The only computer I had access to at the time was a Wicat Systems 68000 with an operating system that was kind of a cross between VMS and Unix. It had a decent C compiler and supported any number of ASCII terminals. Having learned C pretty well in college, I was comfortable building any kind of simple application I needed. So as the business grew, I began tracking things like inventory and production runs in simple flat files with command-line driven programs to manipulate the data.

My idea of a database at the time was the Unix password file. However, I didn't really care for the colon as a field delimiter so I started using a pipe character since it didn't ever occur in my data. Eventually I had a whole suite of programs for dealing with various aspects of the business. I could track my customers, my employees and my vendors. I could also write A/P checks and payroll checks, and I had an accurate bank account balance and a pretty elegant purchasing system.

By about 1993, I was running into trouble with my hardware. Wicat had gone out of business and I literally had a shed full of their old computers to keep my systems running. But I could see that it wouldn't last forever and so I was looking for a more permanent platform I could migrate to. I was not at all impressed with the Microsoft platform and it was clear that a port to DOS would be a lot more work than a port to Unix. So I was getting ready to bite the bullet and buy SCO Unix when I first heard about Linux. I knew instantly it was for me and set forth to get my first system up and running.

I think it only took me a couple of weeks to get my whole system ported over to my first slackware box running a pre-1.0 kernel. My old command-line utilities worked very well on Linux. And with the help of a terminal server, my old Wicat computers became dumb terminals. And as they had problems or as I needed to scale, I could buy new Wyse terminals to keep me going.

Life was pretty good until I hit my next scaling hurdle. In about 1998, it was becoming clear that the business was growing out of my system. I had multiple salesman who wantied to be able to access customer files at the same time. My file locking prevented them from overwriting each other, but it also prevented simultaneous access of various parts of the data. In addition, there was increasing pressure to expand the system to include better accounting and more elaborate tracking of various processes in the business. People were also becoming more accustomed to graphical environments and it was getting harder and harder to get people to be productive with the command-line interfaces. I knew it was time to change again.

Up until that time, I had been of the opinion that "Any program worth anything could and should be written in C." My experience with interpreted languages was limited to Basic and Forth which I considered to be toys--certainly not for use in serious projects. I had heard of databases but didn't really know much about them. I had seen and used graphical programs, but it seemed like an unimaginably complex job to write one so I wasn't sure where to start.

My research on databases lead me to Postgresql. It seemed like the best choice for several reasons: It had early support for triggers, stored procedures and transactional integrity, and it had a good developer community and was working toward compliance with published standards. So I set forth to learn SQL and database architecture.

At the same time, I wanted to learn how to make GUI applications. My research on this lead me to Tcl/Tk. At that time, I didn't find much to compete with Tk for a graphical toolkit. Since it seemed to be inseparably linked to Tcl, I set forth to learn how to use this new (and strange) programming language.

I remember reading for several hours trying to figure out what in the heck a "widget" was. But eventually, it started coming together. And before long I had a simple customer database running with a Tcl/Tk front end and a Postgresql backend.

First pass

I eventually came to call this first attempt my "pass one" version. My Tcl (and SQL) programming was somewhat awkward at that point. For me, Tcl was just the language I had to use in order to get Tk--which is what I really needed to put my graphical widgets on the screen. But an interesting thing began to happen: The more I used Tcl, the more I began to understand the genius of its simplicity. Over time it became the language of choice for many type of tasks.

Specifically, I came to understand the benefits of an interpreted language. Previously I had put a lot of stock in the processing speed of a compiled C program. After all, isn't it better to have a program execute in 100 us, rather than 100 ms? Well, eventually I determined that it isn't always that important. Really, if the difficulty of the program development cycle gets reduced to the point that the application can come into existence at all, I don't really care how fast it executes as long as it is fast enough to keep my employees productive. I could buy faster computers. But I could only code so fast. And I had begun to see how much more I could do and how much faster I could turn it out in Tcl/Tk than I had been able to do in C. I recognized that it wasn't always pretty and not always something I could have readily sold to others. But it was doing the job for my company and we were beginning to see productivity benefits from the new programs.

Also a part of pass one was figuring out how to deal with the objects I was creating in the backend. In Postgres, it wasn't always easy to modify the database (at least for a beginner). And besides that, I just felt uneasy about the idea of issuing create commands and then later issuing alter commands to modify things in place. I didn't really trust the idea of the newly altered version of my database existing only inside the black box of Postgres. I had come from the world of C programming where there was source code, and there was object code. Source code was the authoritative description of the program. Object code was just an instantiation of the real program for this or that target machine. But the object code could be deleted at any time and re-

created as needed from the authoritative source document.

Issuing alter commands to the database felt kind of like using a binary editor to change your compiled object code. What would be the point of that? The next time the program would be compiled, the changes would be lost. Dumping the altered schema from the database felt to me more like running a dis-assembler--reverse engineering at best. I wanted a way to author my schema outside of Postgres in an authoritative source document. And then I wanted to be able to instantiate that schema inside the database any time (and as many times as) I wanted.

That was the beginnings of what would eventually be called Wyseman (WyattERP Schema Manager). At that point, it was a collection of text files and shell scripts. The main concept was simple though: I created chunks of SQL code capable of creating (and destroying) each of the objects (tables, views, functions, etc.) I wanted in the database. Then, I recorded which objects were dependent upon which other objects. Using my scripts, I could then remove any object from the database, and rebuild it fresh from my source documents. If the object had other dependent objects, those objects could be included in the process as well. If the object list contained tables, the data from those tables would first be saved out to files. Then, once the fresh schema components had been created, the data would be imported back in.

Second pass

By the time I had a bare cores of modules running for the business, I had began to become disgusted with my own programming style. I had started to figure out how to make Tcl more modular and object oriented by using namespaces more effectively and by structuring my code better. I had begun to understand the concept of building up more complex GUI components (mega-widgets). And I was getting better with hiding complexity in libraries so my main program could become simpler and cleaner.

Pass two was my next step to implement these changes. Unfortunately, the new structure was totally incompatible with my pass one programs. But as I began to port each application to the new structure, most of my reusable code got tucked into a main library. And my applications got much shorter and cleaner. Often a fairly complex application like a customer contact manager could be expressed in about 150 lines of code. All the rest was now in a library--available for use by other applications.

I began to standardize the way my widgets and mega-widgets would operate. This was modeled after the way basic widgets behave in Tk itself. Each widget was implemented in its own namespace. Each one had a constructor. And when a new widget was instantiated using that constructor, it would always create a widget command for the new instance (a global command with the same name as the widget instance itself) which could be used to access all the functionality in the widget. Each new widget module had "class variables" and "instance variables" much like you would expect in a C++ object. But these were not managed by an OOP language--just by discipline and convention in the coding style.

Tk widgets allowed a suite of command line parameters and switches which could optionally be abbreviated. So I adopted this same structure in my widgets. I added a further extension of this concept which I called "Dynamic Lists." A dynamic list looks just like a set of command line

arguments such as: “-switch1 value1 -switch2 value2 -switch3 value3” except in certain instances, some of the switches can be omitted from commonly used parameters such as: “value1 value2 value3” This way, common parameters could be given in a pre-determined order as shown above. Or all values could be prefixed with a named switch and then given in any order. Additionally, switches could be spacified more than once on the command line. For items that can only hold a single value, the last occurring switch on the line would take presidence. For some items, the system could collect and use all specified values.

Also using dynamic lists, my megawidgets could strip the values out of the command line that they wanted to use. All other switches could remain in argv and simply be passed down to subordinate widgets. So for example if I had a mega version of an entry widget that also included a label widget, my megawidget could strip off the command line information about how to build the label. But things like -background and -length (native to an entry) could just get passed down to the entry widget itself. This made the megawidget appear to inherit all the characteristics of the component widgets it used internally.

Another important part of pass two was the introduction of Wyseman. I had wrestled with the way I was managing my schema objects using text files and shell scripts. I liked the concept of maintaining an external and authoritative source document for the creation of the database schema. I dabbled with the idea of a completely GUI front-end for creating database objects, but it seemed like that had already been done in a number of different ways. I experimented with Filemaker for a time, but I found it very limiting. After all, I wasn't trying to make database design accessible for less experienced users. I was trying to bring better organization and management to potentially very complex schemas. I determined that an abstraction layer between me and the database would only limit the functionality I would be able to access in the abstraction layer. I needed something that would allow me to continue to access every obscure feature Postgresql was able to offer. And while helping with keeping my database objects documented and organized, I also wanted a way to hide the complexity of some of my more elaborate objects such as a macro processor.

Finally, I needed a data dictionary for my objects. In pass one, things like column titles and pop-up context helps were all over the place. I wanted to be able to create and document the objects one time and in one place and then relieve the application of the burden of supplying that information.

I really tried to not do Wyseman in Tcl. After all, I was starting to like Tcl--a lot. I was worried that I had just substituted one dogmatic approach (all programs should be written in C) for another new one (all programs should be written in Tcl). Having previously written a full-on macro language entirely in m4, I experimented with that. I re-considered doing it in C. I tried XML. I tried creating a database schema and storing my sql chunks inside tables in the database (cool from a purist point of view, but introduced a nasty bootstrap dilemma).

In the end, I came back home to Tcl. I determined that my dynamic list format was probably the best and cleanest structure for storing my schema data. The Tcl syntax turned out to be pretty good for storing SQL chunks. Once inside a set of literal quotes ({}) I could express pretty much anything SQL needed, without having to further quoting or escaping. I wrote a fairly simple macro scanner to look inside quoted SQL for escapes back into the TCL interpretor and viola! I

had macro capacity.

The fact that I was writing in native Tcl and storing chunks of native SQL meant that I had preserved the full power of both languages. Instead of being limited to what I could express just in SQL, I could also write Tcl procedures capable of churning out etremely complex SQL objects (like views with long lists of columns and/or rules). I could hide the complexity and messyness of repetitive tasks (like implementing insert and update rules on views).

I did a one-time port of all my object descriptions into the new format, rebuilt the database from my new objects, and reimported my old data. It worked pretty much flawlessly. And I never turned back or regretted the decision to implement Wyseman in Tcl.

Third pass - Wylib

My borderline ADD never seemed to allow me to fully port all of my applications from one pass to the next. By the time I had finished porting about 80% of my applications to pass two, it was becoming obvious what I needed to do next. And I was more anxious to get onto it than to sit around porting really obsolete code to newly obsoleted code. I was ready for pass three.

I was starting to feel like my stuff might be good enough for other people to start using it. Grateful for what open source had done for me, I determined to release something under an open source license. I had already come up with the idea for the name WyattERP but then I found out that some AS400 people were already using it. But it looked like their project was slowing dying, so I kept the name and registered the domains myself.

I began by taking all my shared code from pass two and pulling out the parts that were specific to the ATI implementation. That core of the code became the central library, Wylib (WyattERP Library). Wylib includes wrappers for all the standard Tk widgets. It also includes all the standard mega widgets and support functions for making a WyattERP database application.

As I stripped out site dependent code, it had to go somewhere. So I came up with the idea of a site library. You can name your site library anything you want--just tell Wylib about it by setting the environment variable WYLIB_SITELIB and it will automatically get loaded if it exists. The site library can include any number of customizations to each of the standard modules in wylib. It can also include any other modules that the site designer needs that are specific to his implementation.

As I continued this porting process, it became obvious that there were some modules that didn't really fit into either Wylib or the site library. For example, we developed interfaces to the FedEx web site and to the asterisk phone server. While these were too application specific to really fit into Wylib, still they could be applicable to multiple sites. And so Waplib (WyattERP Application Library) was born.

I dutifully posted early versions of Wylib on the WyattERP website complete with some sample applications. But the demands of the business did not leave me with much time to maintain the distribution. There were many things about the ATI implementation that were proprietary to the business model and so could not really be open sourced without compromising ATI's competitive edge in its market. So I found that I really had to concentrate on keeping the ATI implementation

moving forward and I wasn't able to keep the open source project and its sample applications current and useable. Because there wasn't much in the way of a useable schema design with the project, I think people couldn't really get the hang of it just by downloading the project. So although I left the project site up, I resigned myself to the fact that I would have to just concentrate on keeping our in-house code moving forward rather than devoting time to the open source project.

The next few years were very challenging for me at ATI. It was time to implement a fully GAAP compliant accounting model in the ERP. This took even more time and so any hope of devoting time to the open source project was delayed even further. But with effort, we were able to complete a fully functioning general ledger with most of the data coming from the standard operations modules.

Next, it became necessary to move the business into a new larger facility. I redesigned the way material management would work and implemented a full inventory control system using WyattERP. That took another year or two to get that fully up and running.

Fourth pass - In Process

As always, the ADD started kicking in again around the time pass three was getting close to completion (but never done). Again, it was time to implement lessons learned from past efforts. In pass three, I had wanted to make WyattERP move useable to other sites. Wylib was a good stab at this for the front-end. Wyseman was also fully functional for multiple sites. The main weakness, however had always been in the schema itself. Although I had released a run-time library for the front end, and a tool for managing the schema, my schema itself had always remained closed and proprietary.

By this time, my job duties at ATI were starting to taper off a little. I had also gradually diversified my holdings to include several other businesses to include a farm and a private loan company. Through the development of the accounting functions for ATI, I had learned a good deal of accounting. A lot of the lessons learned were after the fact, so there were a number of things I would have done differently. But I had not yet had the energy to create ERP's for these new businesses. And they were simple enough that I could track them pretty effectively in Gnucash.

But as time has gone by and they are getting more and more complex, I have been feeling more need to get them set up on a real ERP.

So one goal of pass four is to create an actual open source schema to go with Wylib and Wyseman. This is now called Wyselib (WyattERP Schema Library). It turns our that Wyseman is pretty good at selectively pulling multiple bits of SQL together from any number of sources. So the idea is to create a set of independent schema modules. Then the site author can pick which modules he can use "out of the box." And which ones he wants to do custom. In some cases, he might be able to use the standard modules and just add a few custom columns.

Ideally, Wyselib would evolve into a basic functioning schema to include all the basic functions of a simple business. For example: employee tracking, payroll, customer tracking, orders,

material management, inventory, vendor tracking, purchase orders, and so forth.

My goal is to create a schema which I can use to run my current businesses--each one as a different site, but sharing all the code possible in both the front end and the back end. Ideally, I would like to incorporate all the lessons learned from the creation of the ATI schema (although many of the implementation details of ATI's business model will still remain proprietary).

Underlying Philosophy

As WyattERP evolved through its various stages of development, several philosophies slowly evolved. Some of these were based on my own beliefs about how to best run a business. Others had more to do with leveraging the strengths of Tcl, SQL and Linux. In many cases, the lessons were learned by first doing it the wrong way and then improving things in later iterations.

Small Main/Configuration File

In the early days of software development, it was not uncommon to distribute source code to the end user. Likely this was a necessity because no developer would be able to perfectly anticipate the varying needs of every end user. Inevitably, users would need to customize their software somehow to better server the unique needs of their business or institution.

But commercial developers didn't really like distributing their source code because it was too hard to maintain a competitive edge when you are showing everyone exactly how you do things in your code. So in order to facilitate closed source development and still maintain the ability to do a certain amount of customization, it became necessary to invent the "configuration file." In this context, I use the term "configuration file" to mean any sort of data or procedural code, modifyable by the end user, that an application might read at startup or while running to tell it how to behave with respect to the end-user's specific needs.

Often site configurations are maintained in a setup file. And the existence of a setup file implies the need for a specified syntax or language in which to express the setup. The existence of a language for the setup file implies the need for a parser to read and interpret the setup file. And the parser as well as the language itself needs to be thoughtfully constructed so that it gives the end user sufficient access to all the complexities of the application necessary to customize the application well enough for each diverse end user's needs.

Experienced programmers will recognize the fact that they often end up in the parser business in order to configure their applications. In fact, Tcl was invented for the very purpose of creating a standardized syntax for configuring applications. The idea was to create a small, portable parser which could be included inside any larger application. This way, the application could be configured with this Toolkit Configuration Language (TCL) and programmers could then concentrate on writing the rest of the application without having to invent a new syntax and parser each time.

The interesting thing is, in order to make a toolkit configuration language that could be used by any application for any purpose, it would certainly need to be powerful. Specifically, it would need to be "Turing Complete" or include conditionals, branching, internal variables and the like.

Essentially, it would need to be a complete programming language--which Tcl turned out to be.

Then, the programmer is faced with an interesting dilemma. Now that the configuration language is a complete programming language, and the end user can express arbitrarily complex procedural and/or data structures in the configuration code, where does the application end and the configuration begin. In essence, the application becomes a library of specialized functions that perform tasks specific to its area of expertise. But it is the site-specific configuration code which, in the end, can control that application telling it how to behave in the end users' environment.

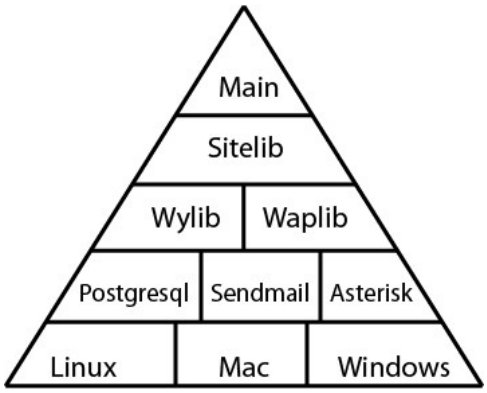
So why is the library the top-level object calling the site-specific code. Shouldn't it be the other way around? At least with open source development, it is easy to do this by turning the configuration paradigm up-side-down.

In this structure the main program, implemented in Tcl, becomes the application. Endowed with a powerful lower level set of library functions, it can call upon those functions in a very high level way to define the operating parameters of the program. If there are two or more applications which operate similarly but on different data or in a different way, ideally the main would contain only that data and those procedures which differ between the two functions. All commonalities would be expressed in shared code and data contained in the shared libraries.

In the ideal case, this makes the main program relatively small and concise (just like an ideal configuration file). All the real work is being done in the shared code. The site specific code (now in the main loop) can access all the richness of the shared libraries, but it is not limited to a pre-conceived set of configuration options. Rather, it also enjoys the full power of the underlying programming language itself. So the programmer can go to any depth necessary in producing application specific code.

Module Pyramid

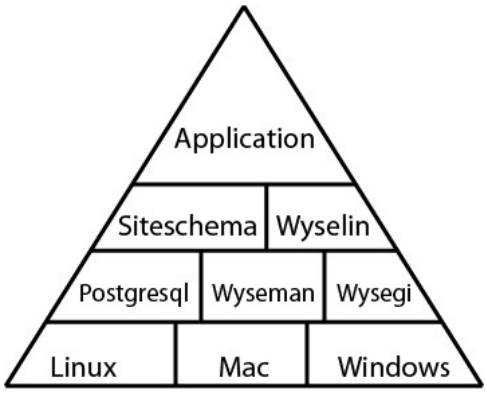
With this code structure, we end up with a pyramid-shaped set of modules or components.



Any time features are needed or added, it is peferrable to add them at the lowest level where they may appropriately belong. That way, code can be shared by as many different processes as

possible.

The front-end code structure above is fairly well organized in pass three and beyond. One purpose for pass four is to apply these principles to the way the back end is constructed. A similar pyramidal diagram can be constructed for the database schema construction:



Exposed Model GUI

Immediately upon using a Wylib-based application, you will notice a very different GUI presentation. This is intentional. Most applications start with the premise that screen real estate is going to be limited. You typically have a single menu bar of some kind across the top, and then a series of virtual screens or pages you can access by pushing the right navigation buttons. Web sites tend to follow this paradigm. You can only view one page at a time so you have to navigate from one page to the next to get your job done.

While there is nothing inherently wrong with this approach, Wylib lends itself better to an approach that might be called a "dashboard" or "control panel" view. The goal here is to get as much data as possible to present on the screen at once and to minimize the amount of navigating that will be necessary in order to do the job. While this approach tends to have a steeper learning curve, my experience is that in an enterprise people can eventually become more productive when the number of mouse clicks can be reduced.

The other important principle is what I call "exposed model." Relational databases are very powerful. The foreign key relationships established in the database tie records together in a way that explains how they are related to each other. For example, we may have one table containing a list of customers. Another table might contain a list of orders.

In the exposed view paradigm, we would try to make it relatively easy to show both the list of customers and the list of orders on the screen simultaneously (not requiring you to navigate between them one at a time). Then if you select a given customer for editing, the list of orders should automatically update to show the orders associated with that customer. In this way, the user will begin to understand the foreign key relationships intuitively--even if they don't know what a foreign key is.

So WyattERP applications will typically have a single menu bar at the top like traditional applications. However it will also contain multiple panes, each of which may have its own menu

bar. And it will be possible to launch other top-level windows for simultaneous display, each of which will have its own menu bar and containing its own panes and their associated menu bars.

In most cases, these new top-level windows express a class of sorts which can be instantiated on the screen any number of times. So in our example, we could have multiple lists of customers showing at once and each one could have its own relational list of orders as well.

Model, View, Controller

Model-view-controller (MVC) is a software architectural method which lends itself well to application development in WyattERP. MVC was not, in the strict sense, in my mind through the early development of the ATI ERP. However, I was very mindful of the need for multiple, simultaneous views accessing a common core of data.

As I studied MVC and attempted to use it in later architecture, I was convinced that it could be helpful in future design phases of WyattERP and Wylib applications. Rather than going into depth in this paper about how MVC works, I would refer the reader to Wikipedia or some other good resource for a detailed explanation. But the basic concept is simple:

The model represents the state of the project or enterprise. It contains all data that expresses how things have progressed so far. And as more progress is made, the model records the new state of the data (and optionally, a record of the changes along the way). So for a business, we will keep a list of our customers and another list of our employees. Each time a customer or employee is added, the data will change and the model will keep track of this state along the way.

It is the job of the model designer to evaluate the real world problem (the enterprise) and determine a way to represent the salient quantities in that reality in tables, rows and columns. Anyone who has designed a relational database before knows that this task can seem deceptively simple until you start trying to pack actual real-world data into your model. So the goal is to find a model sufficiently complex that it can represent the bulk of your real-world cases while being sufficiently simple that your users can effectively interact with it.

The controller portion in the MVC method contains rules about how data changes in the model. In Postgresql and similar products, triggers and rules can certainly constitute a part of the controller layer. For example when we push a button in our GUI to issue a payroll check to one of our employees, there will typically be a set of procedural functions that will have to be performed before allowing the data to be recorded. We might first check to see that valid work time has been recorded by the employee. We might check to see if overtime needs to be paid or if withholdings need to be made. These are all examples of rules or restrictions on how and when the model is allowed to change.

Since WyattERP is based on the client-server model of Postgresql, a lot of our controller code is bound to be running on the back end. However, it is usually awkward to have all control logic happening in the back end. For example, when adding a new employee record, it is usually a good idea to make sure the user has included a birthdate and a valid taxpayer ID number. Ideally, we would disallow the record addition until these are specified.

While this type of a data check can (and should) be performed in the back end, it should

probably also be screened in the front end as well where we are more likely to be able to issue a more user-friendly and specific error message. So in WyattERP, the controller layer is expected to be as much as possible in the back end, but understood to also occupy some space in the front end.

The view consists of the widgets shown on the screen and the buttons or links the user might click in order to accomplish the desired tasks. Ideally, any number of views can exist at one time. And when the user performs an action on the database, he should access the same set of control code regardless of which view he is in. And he should operate consistently and transactionally on the same set of data in the model regardless of which view he is using. Furthermore, when the data changes in the model, all applicable views should ideally update to reflect the change.

Permission Model

When WyattERP is used in a multiple user environment, there will typically need to be some kind of restrictions on which data can be viewed and modified by which users. For example, in a business environment you might want one person to deposit the checks and a totally different person to apply the income to invoices. One person might enter new employees in the database while a different person might issue payroll checks. In this way, there are greater controls on the data and less chance for someone to do something they shouldn't.

WyattERP starts by creating a basic permission model using Wyseman. When schema components are created, you can specify modules (essentially groups or roles) that will be able to access those objects in various ways. Each module permission is created with three different levels: limit, user, and super. So if I create a table and tell it that the "entim" (entity information manager) module will be accessing it, Wyseman will create three roles: entim_limit, entim_user, and entim_super. Additionally, you define which types of accesses each of the three levels will get to the object. For example entim_limit might have select only permission to the table. We might give entim_user the ability to insert and update. But we might only give entim_super delete permissions.

Wyseman allows for a fairly compact syntax to represent which modules and levels get what access to which objects in the database. If your schema includes a user table, you can use Waplib functions to add and revoke permissions to the modules you have defined.

Additionaly, there is support for role groups (like "finance" or "sales" for example, each of which can contain module permissions of various levels. This will allow you to grant a set of module permissions to a user based on their job description.

In addition to granting select, insert, update and delete to database tables, it is sometimes desirable to limit a user's access to certain columns within a table. While Postgresql has support for column-level permissions, WyattERP lends itself well to simply creating a custom view for each module. When defining the view, you can specify separately which fields can be selected, inserted, and updated. Wyselib contains helpful macro functions to help you manage this without creating a large code footprint.

Other fine grained permission controls might include such things as limiting access to a table

based on a location or the time of day. These can easily be handled by rules and triggers within the database.

Main Components

Wlib - WyattERP Library

Wylib contains Tcl and Tk functions which are considered to be site independent and also generic enough that they would be likely to be shared by a wide variety of applications. Wylib is designed to be used with or without Tk. So support and maintenance scripts can require it as well as GUI applications. If you don't want the GUI components, just don't invoke them.

Some of the more common GUI components will be introduced here:

Dbp (Database Preview):

This widget can display all or some of the records from a single table in the database. It looks somwhat like a spreadsheet, showing rows and columns of data. Controls are fairly standard to a multi-column listbox you might see in another application.

This widget is built upon a similar but lower-level widget called an MLB (multi-listbox) which does most of the same things, but is not associated with a database. Rather, it can contain any abitrary data.

Typically, you would use the standard LoadBy button in the menu bar of the Dbp to select a certain class of records you are interested in. Then you would double click on one of the records to "execute" (do something interesting with) it. The Dbp can be operated in a mode where multiple records can be executed at once if wanted.

It is very common to link a Dbp to a Dbe widget (see below). In this case, executing a record will load it into the Dbe.

A Dbp will largely configure itself from the data dictionary created by Wyseman. However, you can specify additional configuration parameters such as the default order in which fields will appear and what action functions will appear.

```
top::add [eval dbp::dbp $w.p -ewidget $w.e {-m clr -m def -m rld -m all -m prv -m sel
-m nxt -m lby -m see -m aex}] entp
pack $w.p -side top -fill both -expand yes
```

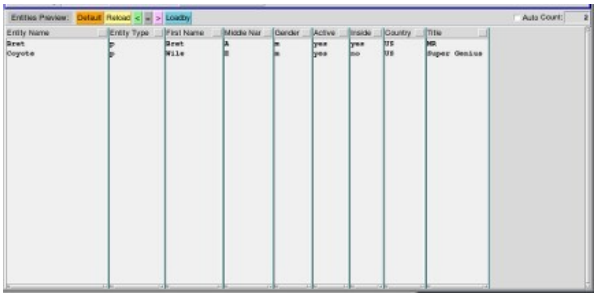


Illustration 1: Standard Dbp Widget

Dbe (Database Edit):

This widget can contain a single record from a single table at one time. Typically, it gets loaded up with a record somehow (often from an associated Dbp). This will allow you to view and edit the contents of the record. Then when you are ready, you can commit your changes back to the database.

A Dbe typically requires a bit more configuration than a Dbp. For example, we need to define where the various fields will appear in the editing pane. While this could probably be done automatically, a human-crafted layout will usually yield a more pleasing result.

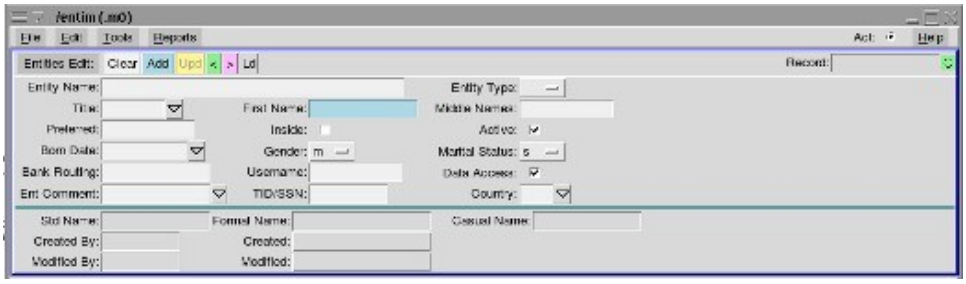
Data Editors:

Wylib contains a wide array of editor widgets for a variety of data values. For example, there is a widget just for selecting dates. Another one is good at selecting the time of day. One displays a listbox with a number of values to choose from. Another will prompt for the input of a number, but displays a calculator for use in coming up with your answer.

These data selection widgets can be specified as data-entry helpers in a Dbe, or as part of a pop-up dialog (or anywhere else for that matter).

```
top::add [eval dbe::dbe $w.e -pwidget $w.p p -table ent_v {-m clr -m
adr -m upr -m dlr -m prv -m rld -m nxt -m {ldr -s Ld} -m sep} -bg blue -
bd 3] ente

pack $w.e -side top -fill both
```



Standard Widget Wrappers

There are a handful of features that I really wished were built into the standard Tk widgets. Absent these, I opted to create a standard wrapper around each one. So Wylib contains classes such as "wentry" which is the Wylib wrapper for an entry. This includes support for a pop-up context help feature, and a way to store the various values that have been used in the entry (a history stack).

Similarly, there is a wrapper around each of the standard Tk widgets. The wrapper gets renamed over the global native widget. So later invocations of a standard "entry" will actually call the wrapped widget, getting the enhanced Wylib entry features.

Utility Tcl Code

Wylib also contains a handful of support functions such as the interface to the Postgresql API. Other examples include support for printing from standard widgets, parsing command line parameters, and interfacing with internet sockets.

Test Suit

While admittedly short on documentation, there is a test folder containing a variety of example scripts to test the various widgets and features. These were typically used during development of the widgets, but can serve as an additional source of useage examples.

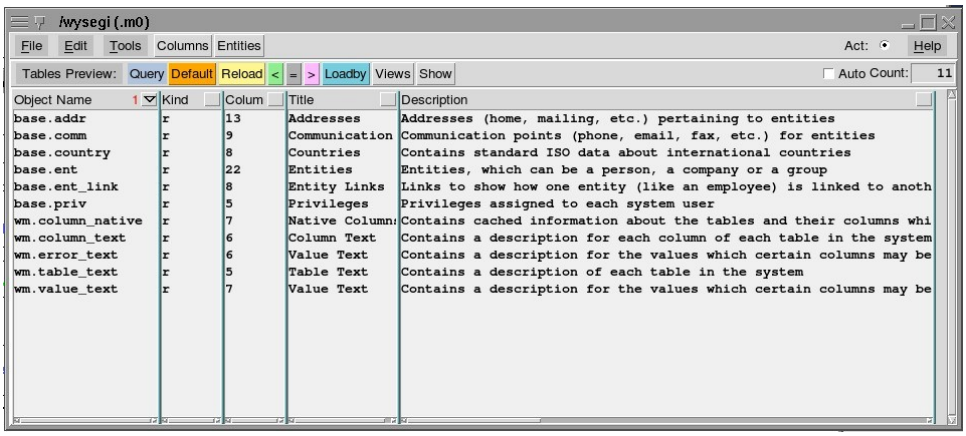
Waplib - WyattERP Application Library

In the early development, Wylib contained everything that was considered "sharable code." But over time, it began to get cluttered with lots of fairly esoteric code that didn't get uesd very often. Waplib became the new home of such code. With the new development of reusable schema components in Wyselib, Waplib also holds the standard front-end support functions to interface to these standardized back-end objects.

Examples of estoteric but sharable code includes a module for interfacing to the Federal Express web site, an a module for creating Nacha (direct deposit) payroll files. It also includes a module for generating paper checks and a graphing module for creating gantt charts (in development).

Wyseman

Wyseman is actually a package that includes a command line tool (wyseman), a graphical interface (wysegi) for examining and changing data in the database, and a run-time library for accessing the data dictionary. When invoked, wysegi will show a list of tables and views in the database.



These database objects are displayed in a standard Wylib preview widget (Dbp) so you have access to all the standard features it offers (such as Loadby, ordering and so forth). If you double click on an object, wysegi will attempt to open a standard Wylib editing widget (Dbe) which will allow you to edit the selected record. These widgets are all automatically generated solely from data available in the data dictionary, so the layout of fields is not as optimized as you might see in a human generated layout.

Most of the time, the database designer should not have to access the data dictionary directly. Rather, the standard Wylib widgets will access it for you in order to display column titles, context helps and so forth. However, it us useful to know what it contains--especially when coding your Wyseman files.

To see the data dictionary, look at the tables contained (using wysegi) in the "wm" schema. For example, you will see a table that holds a titles and helps for each table in the system. Another table holds similar descriptions for table columns. A third table holds values for certain enumerated-value columns you will create.

As you create your own database design, you will author files (described below) that will define the database objects, will populate the data dictionary, and will define how various object data will be displayed on the screen.

At the heart of Wyseman is the wyseman command line utility itself. It will parse your schema description files. And according to how you invoke it, it can build some or all of your schema (your collection of database objects). Or it can destroy (and optionally rebuild) specified portions of it as well.

The Wyseman parser (just the Tcl interpretor itself) understands the following new commands which mirror their SQL counterparts:

- table
- view
- sequence
- index
- function

- rule
- schema

Each of these objects can be created by invoking the command, followed by a Wylib dynamic list of parameters which include:

- Name <object name>
- create <create script>
- drop <drop script>
- grant <module permissions>
- version <object version>
- text <title description>

With the first 5 of these able to be expressed with their switches omitted as long as they are in this exact order. So, for example, we might create a table with the following code:

```
table base.ent_link {base.ent} {
        org_id      int4   references base.ent (ent_id) on update cascade
        , mem_id    int4   references base.ent (ent_id) on update cascade on delete
cascade
        , primary key (org_id, mem_id)
        , role       varchar
        , supr_path   int[]
        subst($glob::stamps)
}
```

Note that we do not have to insert the words "create table <name>" in the create script. We can just start enumerating the column creation portion of the syntax. Wyseman will fill in the blanks if it thinks it needs to .But if it see the words "create table" at the beginning of your script, it will know not to try.

Likewise, there is no drop script specified. Unless there is something fancy involved in dropping this object, just let Wyseman create that part for you. For a table it is just "drop table <name>" so it can figure that out just fine.

Most SQL objects behave this way, you can take certain shortcuts as long as you fill in the parts that can't be figured out. For example, if you create a trigger that calls a function, Wyseman will include the function name in the object dependency list for you (if you haven't done it already).

The dependency list should just be a list of all the object names in the database that must exist before you can create this object. So if you create a view that is based on a table and a function, you should list that table name and function name in the dependency list as shown in the example:

```
view base.ent_link_v {base.ent_link base.ent_v} {
        select eval(fld_list $base::ent_link_se el)
    , oe.name               as org_name
    , me.name               as mem_name
    , el. oid as _oid
        from       base.ent_link   el
        join       base.ent_v      oe on oe.ent_id = el.org_id
        join       base.ent_v      me on me.ent_id = el.mem_id;

eval(rule_insert base.ent_link_v base.ent_link $base::ent_link_v_in {} $glob::stampin)
eval(rule_update base.ent_link_v base.ent_link $base::ent_link_v_up
$base::ent_link_pk {} $glob::stampup)
eval(rule_delete base.ent_link_v base.ent_link $base::ent_link_pk)
} -grant {
   {entim     s {i u d}}
}
```

The grant parameter simply specifies a list including a module permission and a sub-list showing the (possibly abbreviated) permissions to allow to users of levels limit, user and super within that permission. Wyseman will create all the necessary roles if they do not already exist in the database. Remember that in Postgresql, objects like tables and views exist only within a specified database However roles, exist across all databases within your current instance. This could cause you some headaches if you are trying to create multiple Wyseman databases within a single instance.

The object name consists of the exact name of the object as you would refer to it in SQL. For functions, this includes the parenthesis and the full parameter list. This is necessary because Postgresql allows you to overload function names (more than one function with the same name). The Postgresql parser does not care if you put spaces between function parameters, but if you put those spaces in the name of your object in Wyseman it will remove them to create the object name. So when you refer to the object as a dependency of another object, you had better specify it with no spaces in the parameter list (see the example below):

```
function {base.priv_role(name,varchar,varchar)}
```

The procedural language plpgsql also allows alias names for parameters in the declaration. These will be stripped out as well when forming the official object name. So make you do so when referring to the object as a dependency.

```
function {equip_logdep(ei int4, se int4, td date)} {equip_items_v equip_dep
equip_caldep(varchar,numeric,numeric,int4,int4)} {              returns boolean
language plpgsql as $$
```

In addition to the standard SQL objects outlined above, there is one "catch-all" object creator called "other." This is just like "table," "view," or any of the other object creators. Except you need to specify a full SQL create and drop script. Not much can be assumed by Wyseman in this case. This command might be used to create a custom type, aggregate, or operator as shown here:

```
function neqnocase(text,text) {} {
```

```
        returns boolean language plpgsql immutable as $$
        begin return upper($1) != upper($2); end;
$$;}
        other neqnocase_o neqnocase(text,text) {
                create operator !=* (leftarg = text,rightarg = text,procedure =
neqnocase, negator = =*);
        } {drop operator !=* (text,text);}
```

There is a command called "tabtext" which is used for defining titles and context helps for tables, columns, and values. For example, to create the text information for the table defined in the example above, we would include the following:

```
tabtext base.ent_link {Entity Links} {Links to show how one entity (like an employee)
is linked to another (like his company)} {
   {org_id          {Organization ID}      {The ID of the organization entity that the
member entity belongs to}}
   {mem_id          {Member ID}            {The ID of the entity that is a member of
the organization}}
   {role            {Member Role}          {The function or job description of the
member within the organization}}
   {supr_path       {Super Chain}          {An ordered list of superiors from the top
down for this member in this organization}}
} -errors {
   {NBP             {Illegal Entity Org}   {A personal entity can not be an organization
(and have member entities)}}
   {PBC             {Illegal Entity Member} {Only personal entities can belong to
company entities}}
}
```

Note that this same chunk of text can optionally be specified as a parameter to the table or view command itself (using the -text switch).

Another command called "tabdef" is used to define the default way in which columns and values will be displayed in the Wylib GUI. This is not really the same kind of data dictionary information provided by the tabtext command in that this is a little more specific to the point that we are using Tk as a front-end. Realistically, the data expressed inside the tabdef command could be parsed by other front-end generating code. But at the moment all Wyseman does with it is it creates a Tcl library which will be required by Wylib (assuming you make it and put in the right place). It will include properly Tcl-formatted argument lists for attachment to the Dbe (and in some cases, Dbp) widgets. I have given some thought to also moving some of this data to the back end, but at the moment that is not being done. A typical tabdef command looks like this:

```
tabdef base.ent_link -focus org_id -fields {
   {org_id          ent    6      {1 1}         -just r}
   {mem_id          ent    6      {1 2}         -just r}
   {role            ent    30     {1 3}         -spf exs}
}
```

When this dynamic list is parsed, it is simply re-formatted as follows:

```
package provide wmd_acme 1.0
namespace eval wmd_acme {
  namespace export base.ent_link
  proc base.ent_link {{tag {_}}} {
    switch $tag {
      {} {return {-focus org_id}}
      org_id          {return {-style ent -size 6 -sub {1 1} -just r}}
      mem_id            {return {-style ent -size 6 -sub {1 2} -just r}}
      role            {return {-style ent -size 30 -sub {1 3} -spf exs}}
      {_} {return {org_id mem_id role}}
    }
  }
}
```

This code can be referenced directly by Wylib to display the column widgets on the screen the way the user wants.

Note that there are some fairly esoteric switches that can be specified in the tabdef columns. Because each widget at each level just strips off the argument that it wants or needs, you should be able to specify any kind of switch here, all the way down to the background color of an entry in the Dbe.

To discover all the various switches that can be specified, one must understand that a Dbe (database editor widget) consists of an Mdew (Multi data editing widget). An Mdew consists of multiple Dew's (Data Editing Widgets). Dew's consist of a data field (an entry, a text box, a checkbox, a menu button, etc.) and a prefixing label. If you study the available options to each of these classes and understand that the capabilities are essentially inherited as you move up to the megawidget, you can quickly discover what options will be available to you. Until then, to get started, just follow some of the examples provided for the various types of data fields.

Finally, the "define" command is included as a macro facility. Remember that you are in native tcl all the while in your schema description file. So you can use all the power of Tcl including set, for, while and so forth. But on occasion, it is nice to have a small macro processor as well.

So you can define a macro using define as follows:

```
define Tquant {case when a = %1 then b else 0 end}
```

This would then be invoked as simply:

```
Tquant(z)
```

Which would expand to:

```
case when a = z then b else 0 end
```

A macro can have any number of parameters which you would refer to in your definition as %1, %2, %3. Or it can have no parameters. However, if it has no parameters, you must still invoke it with empty parenthesis as follows:

```
define myDef()    1234
```

```
field    int     not null default myDef(),
```

In the sample schema provided, normal database objects are defined in a file with a ".wms" extension. All the tabtext stuff is in a file with a ".wmt" extension. And all the tabdef items are in a file with a ".wmd" extension.

While this is not a strict requirement of Wyseman, it does make it easier to specify which objects you want when it comes to build time. Wyseman command line commands can get pretty complex so I usually use a Makefile to do most of the work. For more complex database designs, there can be a lot of different files to pull object definitions out of. When I start including Wyselib schema components, it can get even more complex.

So I now create a little Tcl script called "modules" in the build directory. This command will produce a list of schema definition files. And optionally, it can find ones that match one or more specified extensions. So for examples, I can specify:

```
./modules wms wmt
```

And it will give me the names of all the *.wms and *.wmt files I need for my schema. I can then call `$(./modules)` from inside my makefile to avoid long lists of files and/or directories where I might be pulling from.

Wyselib - WyattERP Schema Library

The last component is Wyselib. When I first developed the ATI schema, it was all closed and proprietary. Only Wylib, Waplib, Wyseman and so forth were open sourced. I did create a couple of small sample schemas for people to play with to get the idea. But I quickly found that I didn't have the time nor the desire to maintain those sample schema.

So in pass four, I am now moving certain core functions into the new Wyselib. The idea is to build up the operating schema for a couple of my other businesses strictly on Wyselib with a very small corpus of site specific code if necessary. Since this stuff can all be open source, it will be something (hopefully) other people can use. And it will be something I have an incentive to maintain.

If it turns out to be good enough, I may want to go back to the ATI schema and back-port sections of the schema to the Wyselib stuff.

So far, wyselib has a set of payroll withholding functions (actually these are being used in the ATI schema now). It has an experimental base module which consists of a common core for tracking users (entities), their addresses and communication points, and what permissions they have. I also have a module for tracking employees which uses a new nifty kind of pseudo table that actually overlays the entity table. This is very experimental, but it looks like it will have some very cool benefits when fully implemented.

Customers and vendors are planned to be implemented similarly to employees. And I am including some basic accounting and asset management structures as well.

Conclusion

As I reflect on the history of WyattERP, I am happy that it has done such good things for Action Target. We have been able to build a very successful company on a home-grown ERP, and grown through all the stages of our development so far. We are able to maintain GAAP accounting, and track millions of dollars of material flow and other transactions, reconciled in a precise and accountable way.

I had grand plans to see WyattERP used more in the open source community. But it has been largely a one-man show. And I have always had my hands full just meeting ATI's needs. So far, it has been difficult to give the support necessary to maintain the package in a way the open source community could benefit from. However, I continue to be open to the idea. I just think it would require a few other parties who were interested in using the system for their own needs and who would be willing to collaborate on the project to fill in the parts that are missing (like documentation, and other refinements).

I have several other hopes for the future as well. WyattERP has never really been configured for web-wide deployment. At the moment, it has to run on a Linux front-end because of a number of dependencies hastily written into the code. In recent years, I have begun to install the hooks which would allow true multi-platform deployment. But some effort would still be required in order to get there.

In order to use WyattERP for my farm, it will be necessary to achieve this in some degree. For example, one use will require a laptop (probably windows) connected via wireless ethernet to track cattle as they are processed in the field. I would like to be able to just use any old laptop, just hit a web page, and be able to download whatever I need in order to get the application started.

Some people have talked about writing a php or flash front-end for WyattERP. But to tell the truth, the thing that is so cool about it is that it is written in Tcl/Tk. The dashboard design paradigm would be quite a lot of work to re-create inside a browser as well.

So of late, I have become convinced that a well designed starkit might be the best answer. I would like to be able to click a link on a web page and have that download a basic starkit. That application would allow me to connect to the Postgresql database and log in with my credentials. Once the system knew who I was and what module permissions I had, it could then determine what applications I needed. I think these applications could then be downloaded right into my starkit by way of starsync or a similar mechanism.

Once the system had all the latest code for the application(s) I needed, I could have a simple dialog pop up to ask what app to run and the user would be ready to go.

This approach would enjoy several benefits. First, the front-end code could run more or less as-is without needing to be ported to another language or platform. All views would look and act the same regardless of whether they were running over the web or in the main office. The deployment model would benefit from all the advantages of a self-updating starkit. The database author could simply worry about deploying the latest code to a central repository and all his users would automatically get the latest stuff each time they connected to the database.

There are quite a few things I would like to do differently and better when it comes to accounting. There has historically been quite a divide between operations software and accounting software. My experiences of recent years have taught me a lot about what accountants, auditors and banks are looking for in a reporting system. My experiences building several successful businesses have taught me a lot about what operations folks need in order to make their businesses work.

All too often businesses are satisfied to let the operations people shop for their own solution and let the accounting people do a different solution. IT is often successful in creating a more-or-less automated export from the operations system to the accounting system. But it is quite rare that the bridge works in both directions. And it is very rare that the data remains completely consistent between the two sides.

I still don't believe that a successful business can just buy its software off the shelf. Most of the innovators I have seen became successful by developing something in-house that allowed their vision for their own business logic to flourish. After all, if you run your business on someone else's ERP, you are really trusting them to write your business logic. And your competitors can replicate your business quite completely simply by buying the same software you bought.

But when innovators find a new way to do things that is more effective than the competition, they need a way to implement that novel logic in their ERP. They need a way to cook up a system that will capitalize on what they have done that is novel and better. WyattERP provides a platform that can allow that rapid development without the distractions from writing all the GUI and database support code from scratch.