Creating Rich Client Applications using Dabo

Ed Leafe
Paul McNett
PyCon 2010Atlanta

Table of Contents

Introduction to Dabo

Dabo is a framework written in Python that enables you to easily create powerful desktop applications. These can range from those that interact heavily with databases to those that are purely user interface. Like any powerful tool, Dabo can be intimidating and confusing when you first start using it, so this session is designed to give you a good working knowledge of what you need to know to use Dabo effectively.

Target Audience for These Sessions

We are assuming that you have experience programming, and have at least a basic understanding of Python. Obviously, the more you know Python, the easier it will be to follow along. We're also assuming that you have Python and Dabo installed on your computer.

Source Code and Online Docs

The source code for the examples we present in this document can be found in our subversion repository at:

```
https://svn.paulmcnett.com/pycon2010
```

Download it to your computer by installing a Subversion client for your operating system (http://subversion.tigris.org) and then:

```
svn checkout https://svn.paulmcnett.com/pycon2010
```

The latest version of this document can be accessed on Google Docs at:

```
http://dabodev.com/pycon_tutorial
```

What is Installed With Dabo

When you download Dabo, you are actually getting 3 things: the '**dabo**' framework module, which is installed like any other 3rd-party Python module; the '**demo**' folder, which contains the DaboDemo app that shows off some of what Dabo can do; and the '**ide**' folder, which contains all of the visual tools. To be honest, we don't really have an actual IDE; the name comes from our initial idealistic goals of creating a full IDE, but the real world requirements of paying bills has prevented that from happening yet. Both the 'demo' and 'ide' folders can be placed anywhere on your disk that is convenient to you; they are not part of the framework, but are actually a set of applications that are written in Dabo!

Dabo Naming and Programming Conventions

We've tried to be as consistent as possible in creating Dabo, so that developers using Dabo can readily learn the framework and use it effectively. Here are some of the conventions you will come across as you use Dabo:

- The 'd' prefix for classes: the base classes in the Dabo framework are all prefixed with a lower-case letter 'd' (e.g., dTextBox, dBizobj, etc.). We've done this to make the framework classes readily distinguishable from derived classes you may create in your application.
- Class names for all other classes begin with a capital letter. E.g.: Wizard, ClassDesigner, etc.
- All attributes and methods begin with a lower-case letter. E.g.: dabo.ui.areYouSure(), self.requery(), self.sortOrder
- Property names begin with a capital letter. E.g.: self.Form, biz.KeyField, self.Caption
- Long names are made legible by using inter-capitalization instead of underscores. E.g.: requeryAllChildren(), self.AutoPopulatePK
- Properties and attributes that begin with a leading underscore are considered "for internal use only" by the class that they belong to; in other words, they are implementation details that should not be accessed from outside of the class in which they are defined.
- No more than one blank line within a method; two blank lines between methods; three blank lines between classes.
- (Heresy Alert!!!) Use tabs for indentation. It Just Makes Sense™.

Another convention is the use of Python properties instead of simple attributes for most things that you will be setting in your code. Properties give lots of control over the implementation of various behaviors, and are also easily discoverable by documentation tools. We've also created the ability to set any properties directly in the constructor of any object by passing the property name as a keyword argument. For example, let's say you were creating a simple label, but wanted to customize how it looked. You could write:

```
lbl = dabo.ui.dLabel(self)
lbl.Caption = "This is some text"
lbl.FontSize = 14
lbl.FontFace = "Tahoma"
lbl.RegID = "theTextLabel"
```

Or you could simply pass all those property values to the constructor:

```
lbl = dabo.ui.dLabel(self, Caption="This is some text",
        FontSize=14, FontFace="Tahoma", RegID="theTextLabel")
```

The results are equivalent, but the second version is cleaner and easier to understand. Both of these use "instance programming" to manipulate the properties. You may also use subclassing to set common properties via the initProperties() method, which will be discussed later on.

The 3-tier Application Design

The Dabo framework is built around the 3-tier application design pattern. This can cause some confusion, especially for people with experience using the MVC pattern, which is prevalent in the web application world. The two patterns can have some overlap, causing the uninitiated to jump to the false conclusion that "Dabo uses the MVC pattern".

The 3-tier pattern separates code into 3 logical layers: UI/presentation, business logic, and data access. The communication between these layers occurs at only a few points; this is to keep the layers as loosely coupled as possible. We'll discuss the way these work together in more detail later, but for now you should know that they follow the Chain of Responsibility design pattern. In this pattern, when an object receives a message, if it "knows" how to handle that message, it does. If not, it "knows" the next object in the chain, and passes the message to that object, which either handles it or passes it on to the next object in the chain. Eventually the message is handled, and the response re-traces the chain back to the original caller.

Here's a common example: you have a button with the caption 'Save' on it; when the user clicks it, you want to save any data that has changed back to the database. But buttons are UI objects; they know nothing about databases. So like all UI controls, it passes the save request to the window (or 'form', in Dabo-speak) it is contained within. The form can't do anything about it, so it passes the request on to the primary business object (or 'bizobj') for the form. The bizobj *does* know something about saving: it knows how to validate the data to make sure that it conforms to any business logic you've coded. If it fails validation, the bizobj raises an exception, which the form catches, and displays an error message to the user. If the data is valid, though, the bizobj passes the save request to the data tier, to an object called a 'cursor'. The cursor knows how to interact with whatever data store you are using, and makes the appropriate insert or update call. If there are any problems with the database write, the cursor raises an exception, which is re-raised by the bizobj, and caught by the form, which informs the user of the problem. However, if the database write succeeds, control returns normally to the bizobj, then back to the form, and finally back to the button.

Does this seem like overkill to you? Perhaps at first glance, but think about it: each object knows how to do one part of the save process, so when you need to change that part, you know where to make your change, and you can do so without breaking the other parts. Here's what each part does:

Button: allows the user to interact with the app
Form: handles requests from controls, and displays any messages to the user

Bizobj: handles business rule validation
Cursor: handles interacting with the database

So if you change databases, the cursor class will change, but everything else works as usual. Likewise, if your business needs change, you can update the bizobj without anything else breaking.

So while it may take a while to get used to "putting things where they belong", in the long run it will make for a much more flexible and robust app. We have written applications with this architecture that have been running for over a decade; as the underlying business changed, so did the bizobj code. When a new user interface was desired, only that code had to be changed. When the dot-com revolution went into full swing, we could write the web front-end against our already production-quality business layer, providing web applications to our clients very quickly, thanks to this design pattern.

Database Connectivity

Dabo works with many database backends; currently we support the following:

- MySQL
- PostgreSQL
- Microsoft SQL Server
- Firebird
- SQLite

SQLite and MySQL have received the most attention during the development of Dabo, while Microsoft SQL Server has received the least. Why don't we focus on Microsoft SQL Server, since this is arguably the richest source of potential Dabo converts? See: Chicken and Egg.

There are two main classes used for connections: dConnectInfo, and dConnection. Typically you create a dConnectInfo object, populate it with the required information, and then call its getConnection() method, which returns an instance of dConnection. You then pass that dConnection object to create a business object, and work with that bizobj, or you can call dConnection.getDaboCursor() to do direct interaction with your database.

Here's a sample session using cursors directly:

```
import dabo
ci = dabo.db.dConnectInfo(Host="dabodev.com",
        DbType="MySQL", Database="webtest",
        User="webuser", PlainTextPassword="foxrocks")
conn = dabo.db.dConnection(ci)
# Get a cursor to work with
crs = conn.getDaboCursor()
# Execute a query
crs.execute("""select * from zipcodes
        where czip like '145%' order by czip """)
print "-" * 55
print "Number of rows:", crs.RowCount
print "-" * 55
# Print out all the cities and their zipcodes
crs.first()
while True:
    try:
        rec = crs.Record
        print crs.RowNumber, rec.ccity, rec.czip
        crs.next()
    except dabo.dException.EndOfFileException:
        # we're done
        print "Done!"
        break
```

Note that we first define the connection info, and then create the connection. This may seem like an unnecessary step, but the dConnectInfo object encapsulates the differences between various backend adapters way of specifying connection parameters. Also, there is a 'PlainTextPassword' field; by default, Dabo assumes that anything passed in a 'Password' parameter is encrypted, and runs the default decrypting on it. So we use 'PlainTextPassword' so that Dabo knows to use the string as-is.

We then iterate through the cursor's rows using the cursor navigation methods first() and next(). (There are also last() and prior() methods, if you were wondering). You can also set the RowNumber property to directly move the current record pointer. The cursor (and bizobj, as you'll see shortly) have a Record object that reflects the data in the current row. You then refer to each column in the result as simple attributes of the Record object.

Cursors are powerful and flexible objects, but in most normal Dabo programming, you never use them directly. You may even forget that they exist, because all data is handled through the bizobj, and it is at the bizobj layer that you do most of your coding. We're showing them here so you understand the purpose of the data access tier, but also in case you need to access a database backend directly for maintenance and such.

Business Objects

Business objects, or 'bizobjs', as they are commonly called, are the glue that binds together the UI elements and the database backend in a 3-tier Dabo application. They hold all of your application's "business logic". But what if you're not writing an app for a business? Well, the term "business logic" is used because there are several layers of logic that go into an application. "Business" in this context is more like "the business end of the mop" and less like "business attire". At the database side, there are referential integrity constraints that ensure that relational data contains valid references, and for the UI there is presentation logic that determines which forms and controls are shown, enabled, etc. Both of these may be "correct", but there is yet another layer of "correct" to consider.

Consider this example: the user wants to create an order for a customer. They enter 3 items, totaling $20.00. The presentation logic would control the display of the order form, how the items are selected, and valid types of data (e.g., numbers only in the text box for 'quantity'). The database would ensure that all non-null fields have values, and that the relations between the customer record and the order items is correct. But what if you have a rule that the minimum order is $50? A $20 order is perfectly valid to display in the UI, and the database can hold those values without problem. This is what we mean by a "business rule", and this logic would be coded in the business object.

There are two main things you need to configure in a bizobj: its data interface, including SQL Builder or direct SQL statements, and validation routines for the data. For the data interface, there are two main properties that need to be set: **DataSource**, which is the name of the underlying table in the database, and **KeyField**, which is the primary key used to update changes back to the database (If you are using compound keys as your PK, you can specify the KeyField as a comma-separated list of key field names). You then need to specify what sort of data you want to get from the database. You can do this by writing the SQL directly, which is a good option for those who are comfortable with SQL, or you can use the SQL Builder methods to specify the parts of the SQL you want to customize. By default, the bizobj will run:

```
select * from <self.DataSource> limit 1000
```

Bizobjs have a **UserSQL** property; if you set this to any non-empty value, this will be used and *all SQL Builder methods will be ignored*. This is a good option if you have a query that you need to optimize for performance, and the SQL returned by the default SQL Builder methods just won't do. It's also useful if you need to take care of database-specific language that is not generally supported.

SQL Builder

The **SQL Builder** logic allows you to specify the various parts of the query that are needed for your bizobj, and the result will be portable across different database backends. For example, lots of people like to write their unit tests against a local SQLite database to eliminate network factors from their tests, but then need to deploy against a real RDBMS for production. When you use the SQL Builder methods, the syntax differences between the different databases will be handled for you automatically, and it will Just Work.

The SQL Builder methods are:

- **addField(exp, alias=None)** – Add the field (and optional alias) to the fields in the select statement.
- **setFieldClause(clause)** – Sets the field clause, erasing anything previously specified.
- **addFrom(exp)** – Adds the specified table to the 'from' clause
- **setFromClause(clause)** – Sets the from clause, erasing anything previously specified.

- **addJoin(tbl, exp, joinType=None)** – Adds a 'join' to the statement. You specify the table to join, and the expression to use for the join. 'joinType' should be one of 'inner', 'outer', 'left', 'right'. Default is 'inner'
- **setJoinClause(clause)** – Sets the join clause, erasing anything previously specified.
- **addGroupBy(exp)** – Adds the expression to the 'group by' clause
- **setGroupByClause(clause)** – Sets the group by clause, erasing anything previously specified.
- **setLimit(val)** – Sets the limit value on the number of returned records. Default=1000. Set this to None to return all records.
- **addOrderBy(exp)** – Adds the expression to the 'order by' clause
- **setOrderByClause(clause)** – Sets the order by clause, erasing anything previously specified.
- **addWhere(exp, comp="and")** – Adds the expression to the 'where' clause. If there is already something in the where clause, the new expression is added with a default of 'and', but you can also specify 'or' if that is what you need.
- **setWhereClause(clause)** – Sets the where clause, erasing anything previously specified.
- **getSQL()** – This will take all the 'pieces' specified by the various SQL clients methods and construct the actual SQL statement to be used.

Note the pattern for these methods: you can either build up each clause piece by piece (the add* methods), or you can set the entire clause at once. For example, these two are equivalent:

```
self.addField("firstname")
self.addField("lastname")
self.addField("phone")
self.addField("address")
```

and:

```
self.setFieldClause("firstname, lastname, phone, address")
```

So let's continue with our simple app, adding a bizobj into the mix.

```
import dabo
ci = dabo.db.dConnectInfo(Host="dabodev.com", DbType="MySQL",
        Database="webtest", User="webuser",
        PlainTextPassword="foxrocks")
conn = dabo.db.dConnection(ci)
# Create the bizobj
biz = dabo.biz.dBizobj(conn)
# These are the two required properties to be set
biz.DataSource = "zipcodes"
biz.KeyField = "iid"
# Add some fields
biz.addField("iid")
biz.addField("ccity")
# Add an alias for this field
biz.addField("czip", "postalcode")
```

```
# Add a WHERE clause
biz.addWhere("czip like '1452%' ")
# Run the query
biz.requery()
print biz.RowCount
# Show the SQL that was run
print "SQL:", biz.LastSQL
# Iterate through the records, and print the city/zip
for rownum in biz.bizIterator():
    rec = biz.Record
    print rownum, rec.ccity, rec.postalcode
```

Bizobjs also have first(), next(), etc., methods for navigation, but it's handy to have a way to process all the rows in the current result set. That's what the bizIterator class does; as we iterate through the records in the bizobj, it moves the pointer as if we called next(), and when we reach the end of the results set, the iteration ends.

Data Validation

There are two kinds of data validation: field-level and row-level. Field-level validation fires when any UI control that is bound to a bizobj tries to "flush", or save its value to the bizobj; this 'flushing' usually occurs when a control loses focus. Field validation is intended to provide immediate feedback for invalid choices. To add field-level validation, add code to the bizobj's **validateField()** method. When a bound control loses focus, this method will be called by the framework, and the method will receive two parameters: the DataField value of the control (the field name), and the current value in the UI control. You can then use those to determine if the field's value is valid; if not, return a string containing the error message to be displayed to the user. By default, the bizobj's **onFieldValidationFailed()** method is called, which displays the error message in the form's status bar, and sets focus back to the failed control. You can override this method to handle validation failures in a manner appropriate for your application. If the value is OK, either return None, or return an empty string.

The more common validation requirement, though, is **record-level validation**. In many cases, you can't evaluate the validity of data without considering all the columns in the record, and since users can change fields in any order they like, there's no point in evaluating their values until the user attempts to save those changes. As part of the save process, the bizobj's **validateRecord()** method will be called for each modified record in the data set. When called, the bizobj RowNumber will be on the modified record, so you can examine its Record object directly. And like the validateField() method, if a non-empty string is returned, the validation fails, and the save is aborted. The form's **notifyUser()** method is then called, which by default displays the error message in a dialog. You can customize this behavior by overriding notifyUser() in your dabo.ui.dForm subclasses.

Bizobj Relations

One of the most common things when working with relational databases are, of course, related tables. Dabo bizobjs make working with related tables easy; you only need to set a few properties in the child bizobj. The one that *must* be set is the **LinkField** property: it tells Dabo which column is the foreign key to the parent table. If that foreign key is linked to the parent's primary key, which is the most common case, you don't need to do anything more; Dabo will figure out the relationship. But if it's linked to a column other than the parent's PK, you need to set the child's **ParentLinkField** property to the name of the related column in the parent. Once those are set, the only other thing you need to do in your code is tell the parent that it has a child. To do this, pass the child to the parent's **addChild()** method, and you're ready to go! Now when you change the current record in the parent, the child will automatically be requeried to fetch the related child records.

There are a few other properties available to help manage relations. For child bizobjs, there is **FillLinkFromParent**, which, when True, automatically fills in the value of the column specified by LinkField with the value of the related column in parent's current record. There are two properties that control automatic record adding: in the parent, **NewChildOnNew** determines whether, when a new parent record is added, a new record is added to each of its child bizobjs. You can control that individually for each child by setting its **NewRecordOnNewParent** property; only if both are True will a new record be automatically added to a child bizobj when a new parent record is created.

DataSets

When a bizobj's **requery()** method is called, it has one of its cursor objects run the SQL, and then create a **DataSet** of the result. A DataSet is simply a tuple of dicts: each record in the returned data is stored in a dict, with the column name as the key, and the column content as the value.

Sometimes it's more efficient to grab a large number of records from the server, and then filter them locally to fit the user's selections. This is easily done in Dabo by calling the bizobj's **filter()** method, which lets you apply a filter to any column without losing the rest of the data. You can apply multiple filters in series, and then remove them one at a time, like an 'undo' function, using the **removeFilter()** method. You can also call **removeFilters()** to remove any and all applied filters to get back to your original data.

If you need to access the current data in a bizobj, simply call its **getDataSet()** method; this will return a copy of the current data. You can optionally limit the data that's returned by passing any of several parameters that will limit the columns returned, the starting row and total number of rows returned.

DataStructure

This property contains a description of the structure of the underlying data being managed by the bizobj. By default, it is automatically populated after a query, getting the information from the dbapi-level information in the bizobj's underlying cursor. But because there is not a 1:1 relationship between data types and Python types, there can be an ambiguous result as both the dbapi layer and the Dabo layer try to convert the data from the database into the appropriate Python type. To avoid this ambiguity, you can explicitly specify the DataStructure for your bizobj. The format for the DataStructure is a tuple, with one element for each field of data. Each field element is a 6-tuple containing the following:

```
0: field alias (str)
1: data type code (str)
2: is this the pk field? (bool)
3: table name (str)
4: field name (str)
5: field scale/precision (int or None)
```

See the AppWizard-generated bizobj code for examples of DataStructure being set explicitly.

VirtualFields

It is frequently useful to have a field in the data displayed to the user that is not actually present in the underlying database table. Probably the most common example that comes to mind is the total cost of a line item: you would probably store the unit price and the number ordered, but not the total; that is easily calculated by multiplying the two. But since a UI control such as a dGrid expects all its data to come from a bizobj.Record object, there isn't otherwise an easy way to handle showing derived or calculated values.

Define such calculations in the bizobj's VirtualFields property, which is a dictionary that maps the virtual field name to the function to call to calculate its value. In the above example, we would define the following:

```python
self.VirtualFields = {"lineTotal": self.calcLineTotal}
```

and then define the following method in that bizobj:

```python
def calcLineTotal(self):
    return self.Record.item_price * self.Record.quantity
```

If we were displaying the items for a customer order in a grid, we would add columns for the item name, sku, price and quantity ordered, and these would be gotten from the database. We would also add a column whose **DataField** property would be set to 'lineTotal', and the calculated value will display for each row in that grid.

We call them VirtualFields (instead of, say, CalculatedFields) because they are always calculated on the fly, when accessed, which means no calculations are ever being stored (well, unless your virtual field callback function stores them for some reason). So they add very little load-time overhead, and virtual UI controls such as dGrid only request the values of fields that are presently on screen, so even if you have 20,000 records in the current bizobj, the virtual field values will only need to be calculated for the 10 records currently shown on the screen.

Because of the on-access calculations, you want to keep your virtual field functions as lightweight as possible. Don't expect to get great performance if the function needs to iterate over several layers of child bizobjs, for example.

Please note that by definition, VirtualField values are read-only.

The UI layer

Dabo has a rich set of UI controls and a robust event model. While Dabo was designed to be able to use any UI toolkit, such as Tkinter, Qt, and wxPython, for the foreseeable future the UI toolkit used by Dabo is wxPython. Realistically speaking, it's a lot of work to wrap the UI classes; We estimate that about 80% of the time spent on developing Dabo has been spent on wrapping wxPython, and keeping up with that moving target is all we've been able to do.

Why don't we just use the UI toolkits directly? Even if we gave up the design goal of making Dabo toolkit-agnostic and just settled on wxPython, we'd still want to wrap their classes. First off, wxPython really shows it's C++ roots, with lots of STUFF_IN_ALL_CAPS that makes code ugly to read; we've gotten rid of those and replaced them with more Pythonic names. Secondly, wxPython requires you to perform several tedious steps to accomplish a single task; we are able to handle those repetitive and pointless tasks for you, making your code cleaner to write and easier to read. Finally, wxPython is inconsistent in how you do things, which makes it necessary to constantly refer to the Help file as you code. For example, it is very common to have a bit of text associated with a control, such as a title on a window, a column header on a grid, the word 'OK' on a button, etc. In wxPython, these three examples each use a different method to set that text, and it can be hard to remember which one uses which method. We've wrapped these differences so that in all three cases, you simply set that object's Caption property, and it does what you expect.

Here is a list of the most common UI properties:

- **Caption**: many controls have a small bit of text displayed on them. Buttons, labels, window title bars, the tabs on tabbed pages, etc. All of these can be set using the Caption property.
- **Value**: while some controls are display only, others, such as text controls, check boxes, list boxes, sliders, etc., all have a value associated with them. The Value can be of any data type.
- **DataSource**, **DataField**: these are the basis of data binding. When these are set, the control gets its Value from the specified DataSource/Field, and changes to the control are propagated back to the DataSource/Field. Typically in a database application, the DataSource is a bizobj, and the DataField is the name of the column in the data set. But a DataSource can be any object, and the DataField can be a property of that object, so that

changes to one control can immediately affect another control. The possibilities are endless!

- **Width**, **Height**, **Size**: these control the actual size of the control when sizers are not in use. Width and Height are integer values, while Size is a 2-tuple of (Width, Height). When sizers are in use, these control the minimum size that the sizer will be allowed to size the control at runtime.
- **Left**, **Right**, **Top**, **Bottom**, **Position** – These control the placing of the control in its parent container. The first 4 are integers, while Position is a 2-tuple of (Left, Top). When using sizers, setting these is pointless, because the sizer will control the positioning of the object; essentially, they are read-only with sizers.
- **Parent**: a reference to the object that contains the control.
- **Form**: the top-level window that contains the control. For a form itself, the Form property returns None.
- **BackColor**, **ForeColor**: the background and foreground color used by the control. These can be set with RGB tuples, such as (255,128,0), or with common color names such as 'blue', 'RED', 'lightGreen'. Note that capitalization is ignored for colors.
- **Children**: returns a list of all objects contained by the control.
- **Enabled**: determines if a control can be interacted with by the user.
- **Visible**: determines whether the control is shown or not.
- **FontBold**, **FontFace**, **FontItalic**, **FontSize**, **FontUnderline**: used to set aspects of the displayed font on the control.
- **Name**: uniquely identifies a control among other controls with the same parent container. All objects have a non-blank Name.
- **RegID**: optional. If set, uniquely identifies an object within a given form, and a reference can be obtained by treating the RegID as an attribute of the form. For example, if a particular control has a RegID="theThing", any other control on the form can refer to it as self.Form.theThing, no matter how deeply buried it is in any containership hierarchy.
- **ToolTipText**: holds the text of the tool tip displayed when the user hovers the mouse over the control.

There are many other properties that can be used, as well as some that are specific to certain types of controls, but these will cover 80% of what you need.

A large portion of the UI properties can also be made "dynamic". A dynamic property is controlled by a function called at the appropriate time by the UI, such as when the object is getting painted, or the data has been requeried. The dynamic properties are simply the regular property's name with 'Dynamic' prepended to it, and instead of setting them to a simple value, you set them to a callable. For example, if you want to make the visibility of a textbox dependent on some condition, you could use DynamicVisible:

```
def myDynamicVisible(self):
    return self.chkMarried.Value

self.txtSpouseName.DynamicVisible = self.myDynamicVisible
```

Now, the textbox for entering the spouse name will only be visible if the person has been checked as 'married'.

General UI Programming Tips

An application with a graphical user interface differs from non-GUI apps in that there is a main event loop that waits for the user to do things, and then processes those events as they happen. Occasionally, though, you may find that the code you write doesn't seem to do what you were expecting: for example, you may write code that fires when a control's value changes, but which depends on that new value being 'visible' by other objects in the system. Due to the order in which events get processed, though, your code may fire before the control's value has been updated, and you get stale values. This is not an error on your part; it's just a common pitfall of event-driven programming.

To work around this, use the **`dabo.ui.callAfter(mthd, *args, **kwargs)`** function to add your method call (and any parameters) to the end of the event queue. This will ensure that any previous processing has already been completed before your method is executed. There is an analogous function for delayed setting of properties: **`dabo.ui.setAfter(obj, propName, value)`**.

Another common problem in a GUI environment is excessive duplicate calls to the same function, such as **refresh()**. There may be several objects that call refresh() when they are modified somehow, and the refresh() code triggers changes in other objects that end up calling refresh() again, and so what happens is that refresh() can be called dozens of times when once would have sufficed. To solve that problem, use **`dabo.ui.callAfterInterval(interval, mthd, *args, **kwargs)`**.; there is an analogous function **`dabo.ui.setAfterInterval(interval, obj, propName, *args, **kwargs)`** for setting properties. Here '**interval**' is the number of milliseconds to wait for duplicate calls; the other arguments are just like those in dabo.ui.callAfter(). The way it works is that when this call is received, Dabo checks an internal log to see if it already has a call pending with the same signature. If not, it creates a timer that will fire in the specified interval, and adds that call to the log. Now lets assume that another call for the same method with the same args is received. Dabo sees that it is a duplicate, so it resets the timer and discards the duplicate requests. If several dozen similar calls are received, they are likewise discarded. Once the duplicates stop long enough for the timer to fire, the method is called and removed from the internal log.

Common UI Objects

- **Forms**: All UI elements exist in a top-level window on the screen; in Dabo, we call these windows 'forms'. Every UI control must be created with some sort of 'container' control as its Parent; the first parameter of every UI object creation statement is that required Parent object. Forms are the only UI object that can have None as a parent, although you can create forms (and dialogs) as children of other forms, so that when the parent form is destroyed, the child form is also destroyed.

- **Panels**: Panels are convenient container objects for grouping related controls. Normally they have no visible appearance, but you can set their BackColor, or give them a border, etc., if you want them to be visible. You can nest panels within each other to any degree you like. Note to Windows users: in order for forms to look 'normal' for the given theme, you must add a panel to the form, and then add all your controls as children of this main panel. Otherwise, the background appears as a dark grey, and your form will not look right. A great use of panels is to save encapsulated designs as classes, so that everywhere you need to get standard contact information from the user, you just instantiate and place your *PnlContactInfo* subclass of dPanel.
- **Buttons**: There are several button types available in Dabo, each with a slightly different usage:
    - **dButton**: A regular button, with a text Caption
    - **dBitmapButton**: Use this when you want an image on the button instead of just text.
    - **dToggleButton**: This is a button with two visually distinct states that toggles between them when clicked. In this respect it works much like a check box, but looks like a button. You can set different images for the normal and pressed states.
    - See also **dHtmlBox**, which doesn't seem like a button at first glance, but it can be set to launch your own code in its onHit event handler.
- **Text Controls**: Depending on your needs, use one of these controls:
    - **dTextBox**: The standard one-line text entry control
    - **dEditBox**: For multi-line text entry.
- **Text Labels**:
    - **dLabel**: The standard way to display text. Labels have properties that enable the text to wrap across multiple lines.
    - **dHyperLink**: Creates a hyperlink on the form that the user can click to have the link open in their default web browser, or to launch some application code, much like a dButton..
    - **dHtmlBox**: This control allows you to use simple HTML markup to easily display formatted text.
- **Paged Controls**: Dabo offers several 'paged' controls; these are containers that have one or more pages, with only one of the pages visible at any given time. The main difference is how the active page is set:
    - **dPageFrame**: each page has a 'tab' with identifying text. Clicking the tab brings that tab's page forward. The position of the tabs is specified at creation with the TabPosition property (default=Top).
    - **dPageList**: presents the captions for the various pages as a list. Selecting one of the captions from the list selectes the associated page.
    - **dPageSelect**: similar to dPageList, but the various page captions are displayed as a dropdown list control. Selecting the caption from the dropdown brings the associated page forward.
    - **dPageFrameNoTabs**: this is a paged control with no user interface for changing the active page. Page selection must be done programmatically, usually in

    response to some user-generated event, such as clicking a “next” button that you've defined.
    - **dPageStyled**: Among other nice styling effects that this control offers, you can also choose not to show the page tab if there's only one page.
- **List Controls**: Dabo offers several controls for displaying a list of items; they differ in how the items are displayed, and how the user selects an item or items. With list controls, you specify the list of items by setting the **Choices** property to a list of the strings to display. You can also set an optional **Keys** property; this is a 1:1 list of key values, each of which is associated with the Choices value in the same list position. You can specify the **Value** of the selected item in several ways. First is the text that is displayed; this is called **StringValue**. Second is the position of the selected item within the list; this is **PositionValue**. Finally, if keys have been set for the control, you can get the key associated with the selection by referencing the **KeyValue** property. There is another property named **ValueMode** that can be one of 'String', 'Position' or 'Key'; how this is set determines which type of value is returned when you reference the **Value** property. The list controls that you can use in Dabo are:
    - **dListBox**: this control displays several items at once, depending on the height of the control. If there are more items than can be displayed at once, scrollbars allow the user to see the hidden items. If MultipleSelect is True, the user can select more than one item at a time by shift-clicking, control-clicking, etc.
    - **dDropdownList**: this control displays a single item at a time, but clicking on the control causes the list of items to be displayed. Once the user selects an item from that list, the list closes and a single item (the selected one) is displayed.
    - **dComboBox**: similar to dDropdownList, except that the user can type in the area where the single item is displayed. Normally you would have to write code to handle what is to be done to the typed value, but if the AppendOnEnter property is True, the text will be added to the list of items in the control if it isn't already present.
    - **dListControl**: this is a cross between dListBox and dGrid. Like a listbox, you can only interact with rows, but like a grid it can have multiple columns in each row.

- **Grid**: Fully describing the **dGrid** control could take an entire chapter – maybe even several chapters. But this will serve as a basic introduction into the **dGrid** class and how to use it effectively. Grids are used to display data records (called a '**dataset**') in a tabular format, similar to a spreadsheet. If you simply have a dataset named 'ds' and want it displayed, you can call `dGrid.buildFromDataSet(ds)`, and a grid will be constructed for you, with a column for each column in the dataset. However, if you want control over the appearance of each column of data, create a **dColumn** instance for each column in the dataset, and set the dColumn's properties accordingly. We will discuss these shortly. Grids have lots of cool behaviors built into them, such as sortable columns, in-place editing, auto-sizing column widths, and incremental searching within a column. You can modify or turn off any of these if you wish, either on a column-by-column basis or grid-wide, but in general they make grids very handy for your users. Generally, you assign the **DataSource** property of the grid to control where the grid gets its data (usually a bizobj), and assign the **DataField** property to each column to control which column in the dataset is displayed in that grid column.

Here are the grid properties most commonly-used to control the grid's appearance:

- **AlternateRowColoring, RowColorEven, RowColorOdd**: when AlternateRowColoring is True, it colors alternate rows of the grid according to the two row color properties. This makes it easier to visually separate the rows of the grid.
- **Children, Columns**: these are aliases that both return a list of the dColumn instances that comprise the grid.
- **CurrentRow, CurrentColumn**: these determine the location of the selected cell of the grid.
- **HeaderBackColor, HeaderForeColor**: these set the default back and forecolor for the column headers. Individual columns can override with their own value, which is the general pattern for all properties in dGrid: set the grid how you want most columns to behave, and then override the properties in specific columns that you want to behave or display differently.
- **NoneDisplay**: this is the text displayed when null values (None) are present.
- **ResizableColumns, ResizableRows, SameSizeRows**: the first two determine if the user can change the width of columns or the height of rows by dragging the separator lines. The last one controls whether changing the height of one row changes all the rows to that height.
- **SelectionBackColor, SelectionForeColor**: These control the color of any selected cells in the grid.
- **ShowColumnLabels, ShowRowLabels**: these determine if the column headers or row labels are displayed.

These are the properties used most commonly to control the grid's behavior:

- **ActivateEditorOnSelect**: when the grid is editable, should the cell go into editing mode as soon as it is selected? Or should it wait to be clicked after being selected in order to display its editor? This property determines the behavior.
- **Editable**, **Searchable**, **Sortable**: these determine, respectively, if anything in the grid can be edited, searched or sorted. If they are False, then that behavior is disabled for the entire grid. However, setting it to True doesn't do anything by iteself; each grid column for which you want that particular behavior must also have its corresponding property set to True before the behavior can occur.
- **HorizontalScrolling**, **VerticalScrolling**: these control if the grid can be scrolled in the respective directions.
- **MovableColumns**: columns can be re-arranged by dragging their headers. Setting this property to False disables this ability.
- **MultipleSelection**: when True, the user can select more than one cell/row/column at a time.
- **SearchDelay**: when running an incremental search, this controls how long a pause must be before the next keypress is considered a new search instead of appending the key to the current search string.

- **SelectionMode**: can be one of either Cells, Row or Column. This determines if clicking on a cell selects just that cell, that cell's row, or that cell's column.

Columns have a long list of their own properties; here are the most commonly-used:

- **Caption**: the text displayed in the Header.
- **Order**: this is an integer that controls the order of the column within the grid. You can give your columns numbers like 1,2,34,... or 10,20,30... or even 1,66,732. The grid will display the columns depending on the relative value of their Order property.
- **Editable, Searchable, Sortable**: See the grid properties of the same name above. When these are enabled for the grid, enabling them for the column allows the behavior to happen.
- **Visible**: this can be toggled to show/hide a column.
- **Width**: this determines the Width of the column.
- **BackColor, ForeColor**: these control the color of the cells in the column.
- **FontBold, FontFace, FontInfo, FontItalic, FontSize, FontUnderline**: these control the appearance of the font for the cells in the column.
- **HeaderBackColor, HeaderFontBold, HeaderFontFace, HeaderFontItalic, HeaderFontSize, HeaderFontUnderline, HeaderForeColor**: like the above, except these control the column header's appearance.
- **HorizontalAlignment, VerticalAlignment, HeaderHorizontalAlignment, HeaderVerticalAlignment**: these determine where in the cell or header, respectively, the text appears. The former can be one of 'Left', 'Center' or 'Right', while the latter can be one of 'Top', 'Middle' or 'Bottom'. Default is Center, Middle.
- **Expand**: when set to True, the column expands/contracts proportionally as the grid is resized.
- **Precision**: determines how many decimal places are displayed when float values are in the column.
- **WordWrap**: when True, long cell contents will be wrapped onto additional lines as long as their is enough height in the row to display those lines.
- **CustomEditorClass, CustomEditors, CustomRendererClass, CustomRenderers**: you can define custom controls to handle how data is rendered and/or edited in a grid. The CustomEditorClass and CustomRendererClass are used when you need data in a column to be treated differently than the default. You can get even finer-grained control by creating a dict with the row number as the key and a custom editor/renderer as the value, and setting the CustomEditors/ CustomRenderers to this dict. Now any row with an entry in this dict will get its own editor/renderer; the rest get the standard CustomEditorClass/ CustomRendererClass.

Exploring the UI Controls

Each of our base UI control classes contains test code that will execute if you run the .py file directly. Some controls just contain a very basic script that creates an instance of the control, while others have more elaborate test cases, but in any event you'll be able to run the control's test and interact with it. If you're running OS X or Linux, **`cd`** to the **`dabo/ui/uiwx`** directory, and type '**`python <classname>.py`**' to run that class's test. E.g.: type '**`python dButton.py`**' to run the dButton test. If you are running Windows, you can either do the same from **`cmd.exe`**, or you can locate the file in Windows Explorer and double-click it to run.

You will see a window open, with (at least) one instance of the control in question. You can then click it, type in it, whatever; most tests contain handlers bound to the most common events that will print information when an event is received. Additionally, you can open a Command Window by typing Control-D (on Macs type ⌘-D); from that interactive shell you can grab a reference to any control using the dabo.ui.getMouseObject() command, and then explore its properties and methods using the command window's built-in code completion.

UI Events

User Interface code is launched in response to events. For example, some common scenarios are the form becoming active, the button being clicked, the menu item being selected, or the timer reaching its interval. Event-driven programming puts the user in control to use features of your application when they choose to do so. Many events are raised every minute of your application's existence, but typically you only care to act upon a few of them. Connecting your code to respond to UI events is called "binding". You bind the raised event to a "callback function", so named because the function is "called back" from the raised event binding at the appropriate time.

Dabo uses its own event layer for two reasons. First, keeping with the design of making it possible to use multiple UI toolkits and not just wxPython, we needed to abstract out the events. Second, even if we were only going to ever use wxPython for our UI layer, the design and naming of events in wxPython is confusing and very C++ -ish. We've simplified the events, and given them simple and consistent names. All Dabo events are defined in the **dabo.dEvents** module.

For example, most UI controls have a common way to interact with them: you click a button, you select from a menu or a list, you increment or decrement a spinner; you toggle a radio button. In wxPython, each of these different events has a different name, but in Dabo we've simplified it so that these actions all raise the **Hit** event.

To bind an action to an event, you have several choices. The most direct is to run the following code:

```
self.bindEvent(dabo.dEvents.Hit, self.handleHit)
```

Assuming that this code in written in the class for a control, it will now call the **handleHit()** method of that class.

We've provided an even easier way to bind to events: simply name your event handler method '**on<EventName>**', and the binding will be done for you. So in the example code above, we could have simply named the event handler method '**onHit**', and we wouldn't need the call to **bindEvent()**! Dabo will take care of the binding for you.

There is one more way to do the binding: you can pass in the event name and handler to the constructor of the object as if it were a property, and Dabo will figure it out. In this case, the event handler is most likely in the class that is creating the control, not in the control class. So let's assume for this example that we were coding a form, and wanted to add a button to it. We wanted to handle button clicks in the form's 'buttonClick' method. The code to add the button would look like:

```
self.button = dabo.ui.dButton(self, Caption="Click Me",
        OnHit=self.buttonClick)
```

Note that in this case, in keeping with the Dabo style that property names begin with capital letters, the convention is '**On<EventName>**' for the keyword name, and the handler as the argument.

Event handlers need to be written to accept a single event object parameter. This event object will have an attribute named '**EventData**' that will contain relevant information about the event. For example, mouse events will include the x and y coordinate of the mouse, whether any modifier keys were pressed at the time, and, if relevant, mouse scroll wheel information. Keyboard events will contain the value of the key that was pressed, and also any modifier keys. Menu events will contain the menu prompt of the selected menu; tree events will contain info about the selected node; grid events will contain row/column values.

There may be times that you want to stop an event. For example, if the user is typing in a textbox and you have a hotkey that will perform some action, you probably don't want the default behavior of having that key added to the textbox contents. In cases like this, calling **evt.stop()** from the event handler will block the default behavior.

Common UI Events

- **Hit**: Nearly every control has an obvious way that the user interacts with it: a button is clicked, a selection is made from a list, a menu item is selected, a timer fires, a chcekbox is toggled, a slider is dragged, etc. Each of these actions by the user raises a Hit event, so in your coding you don't need to worry about what each control calls its main event; you know that Hit will be the one you want to bind to.
- **GotFocus**, **LostFocus**: these are raised when a control gets focus and loses focus, respectively.

- **KeyDown**, **KeyUp**, **KeyChar**: these events allow you to respond to key presses. Important EventData keys are: keyChar, keyCode, unicodeChar, unicodeKey, controlDown, shiftDown, commandDown, altDown, metaDown.
- **MouseLeftDown**, **MouseLeftUp**, **MouseLeftClick**, **MouseLeftDoubleClick**: note that there are versions of these methods for the right and middle mouse buttons; these are named the same, but with 'Right' and 'Middle' replacing 'Left' in the event name. Important EventData keys are: mousePosition, dragging, controlDown, shiftDown, commandDown, altDown, metaDown.
- **ContextMenu**: this is raised by either a right click or when the context menu button is pressed. The same EventData keys for the mouse events apply here.
- **Activate**, **Deactivate**, **Close**: these events are raised when the form or application becomes frontmost, loses frontmost status, and is closed, respectively.
- **Update**: an Update event is raised by the framework when data needs to be refreshed.
- **Resize**: raised when an object is resized.
- **Destroy**: raised when an object is released.
- **Idle**: raised once after all other events have been processed by the object. It communicates that the object is ready and awaiting new orders.

Developing With Dabo

Creating a Simple Application

OK, enough theory and background for now. Let's start by creating the most basic app. We will be using the command line and simple text scripts for now; later on we will use the visual tools to make a lot of this easier. So for each example script, copy it into a plain text file, give it a name like 'sample.py', and run it by entering the command 'python sample.py' at the command line.

This is about as simple a script as you can get:

```
import dabo
app = dabo.dApp()
app.start()
```

When you run this, you get a blank form displayed, along with a basic menu. Close the form, and the application exits. Now let's replace that boring blank form with something a little more interesting:

```
import dabo
dabo.ui.loadUI("wx")

class HelloPyConForm(dabo.ui.dForm):
    def afterInit(self):
        self.Caption = "Hello PyCon Form"
        self.lblHello = dabo.ui.dLabel(self,
                Caption="Hello PyCon", FontSize=24,
                ForeColor="blue")

app = dabo.dApp()
app.MainFormClass = HelloPyConForm
app.start()
```

Here we defined a form class that inherits from the base Dabo form class: **dForm**. All the base UI classes are in the **dabo.ui** module. Since Dabo is designed to support multiple UI toolkits, we need to tell Dabo which one we want to use, so we add the command **dabo.ui.loadUI("wx")** to load the wxPython toolkit into the dabo.ui namespace. **dabo.ui.loadUI()** must be called before subclassing any of the UI classes.

We customize the form in its **afterInit()** method. This method is automatically called by the framework for every object in order to provide you with a place to do this sort of customization without having to override (and potentially mess up) the native **__init__()** method. There is another useful 'after' method available, named **afterInitAll()**. The difference between the two is that afterInitAll() is called after the object and all contained objects have been created. So in a form with several objects, there is a distinct order in which these various things happen and when the corresponding methods are called. To illustrate this, let's imagine a slightly more complex example: a form that contains an editbox, a panel, and a textbox. The panel contains a label. Here is the order that the afterInit() and afterInitAll() methods are called for each of these objects when this form is created:

1. form creation
2. form.afterInit()
    a. editbox creation
    b. editbox afterInit()
    c. editbox afterInitAll()
    d. panel creation
    e. panel afterInit()
        i. panel label creation
        ii. panel label afterInit()
    f. panel afterInitAll()
    g. textbox creation
    h. textbox afterInit()
    i. textbox afterInitAll()
3. form.afterInitAll()

The basic structure of an application is to define the MainFormClass for the app, and start the event loop by calling app.start(). You can also subclass the dApp class to add your own custom behaviors, and then reference them from anywhere in your app using the `obj.Application` reference that's available to all objects in Dabo.

Ready for some really cool stuff? Run the 'Hello PyCon' app again, and then from the File menu, select '**Command Window**'. You'll see a new window pop up; it will most likely be too small to work with, so resize it to something more reasonable, and position it so that you can see the main form, too. Dabo will remember the size and position of your various windows when you close them, so that the next time you run the app, the windows appear as you left them.

The command window is an interactive Python interpreter that you can use to investigate and modify your running apps. To make it easy to use, we've mapped the name 'self' behind the scenes to refer to the form that was frontmost when you brought up the command window. You can test this out by typing `self.Caption` and pressing Enter; on the output frame on the bottom you should see 'Hello PyCon' displayed. You will also have noticed the code completion popup as soon as you typed the period after 'self': all the known attributes, methods and properties of the object referred to by 'self' (in this case, the 'Hello PyCon' form) are displayed, and as you type successive letters of the desired property ('Caption', in this case), the selected item in the list changes to match what you've typed. Once you have the one you want selected, press Enter to have it entered into the command for you.

You can also change things on the form, too. To see this, press Control-UpArrow once to go to the previous command, and then type at the end to modify the command so that it reads: `self.Caption = "I changed this"`, and press Enter to execute that command. If you look at the form, its caption has now been changed interactively.

Let's not stop here; let's create a control on the fly! Type this command:

```
dabo.ui.dButton(self, Caption="Click Me")
```

You should see the new button appear, but unfortunately it's right over the label. To make matters worse, we didn't save a reference to the button, so how can we move it? Well, there's the **Children** property of the form, but an even handier way is to call the UI method getMouseObject(), and this will return a reference to whatever Dabo object is below the mouse cursor when the command is executed. So place your mouse cursor over the new button without clicking in the 'Hello PyCon' form, and in the command window type:

```
btn = dabo.ui.getMouseObject()
```

Press Enter, and now 'btn' is a reference to the new button. And in the future, try to remember to assign a local reference to newly created objects. For example:

```
btn = dabo.ui.dButton(self, Caption="Click Me")
```

Note to Windows users: some versions of Windows act odd with the getMouseObject() command, moving the command window to the bottom of the stack while still displaying it normally. Just click the button for it in the task bar, and it will return to normal.

Now that we have the reference, we can play with the button. Try entering commands like this:

```
btn.Left = 50
btn.Bottom = 150
```

...and you should see the button moving as the commands execute. Of course, if you click the button, it looks like it's being clicked, but it doesn't do anything. Let's create a simple method on the fly and bind it to the button's Hit event, which is fired when the button is clicked. Type the following code in the command window:

```
def clicker(evt):
    import time
    self.Caption = time.ctime()

btn.bindEvent(dabo.dEvents.Hit, clicker)
```

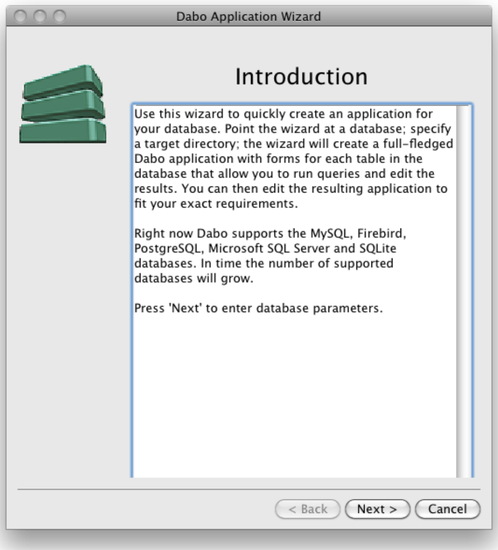
Now when you click the button, the Hit event will be raised, and since it is bound to the clicker() method, that method fires. All event handlers receive an event object parameter; while we don't use it here, we still have to accept it. And since this method was defined in the Command Window, 'self' refers to the form. So each time you click the button, the Caption of the form will change to the current time!

This is still pretty basic stuff; you can make much more complex forms by adding more and more controls. Getting them to all appear nicely will require that you specify the size and position of each explicitly, or use sizers to do all that for you. Sizers are a whole topic unto themselves, and we'll cover them later. For now, we cannot stress enough that although they may seem difficult to grasp at first, the investment in learning how to use sizers will pay off greatly in the long run. They handle issues such as screen resolution, font size changes, OS variations, and form resizing for you, so that your app always looks "right". Or, at the very least, like a clock that is ticking but 20 minutes fast, "consistently wrong".

The Application Wizard

Dabo has a tool called the **AppWizard** that will create a very functional CRUD (Create, Read, Update, Delete) application in about 30 seconds. It is ideal for basic table maintenance applications, but it is certainly not limited to the basics: you can extend your app to work just about any way you want. Some people like starting their development with the output of AppWizard; it gets them started with a functional app, which they can then customize.

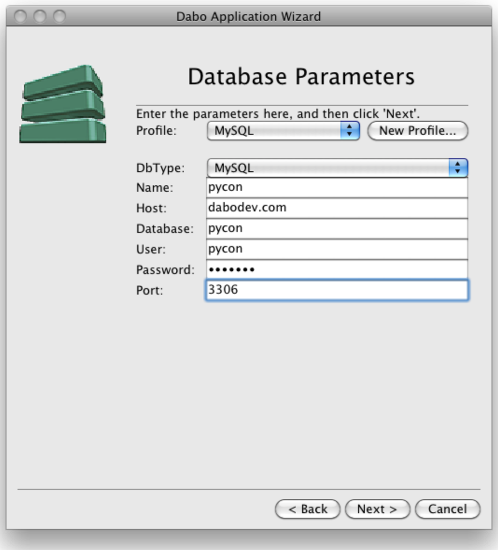
To create an AppWizard app, run **`python <ide path>/wizards/AppWizard/AppWizard.py`**, where <ide path> is the path to your Dabo 'ide' directory. You will see a screen like the one below left. Click the 'Next' button, and you get the screen to enter your database connection parameters (below right). For this PyCon demo, I've set up a public database on the dabodev.com server named 'pycon', and you can connect with username='pycon' and password='atlanta'. Once you have that information entered, click 'Next'.



Use this wizard to quickly create an application for your database. Point the wizard at a database; specify a target directory; the wizard will create a full-fledged Dabo application with forms for each table in the database that allow you to run queries and edit the results. You can then edit the resulting application to fit your exact requirements.

Right now Dabo supports the MySQL, Firebird, PostgreSQL, Microsoft SQL Server and SQLite databases. In time the number of supported databases will grow.

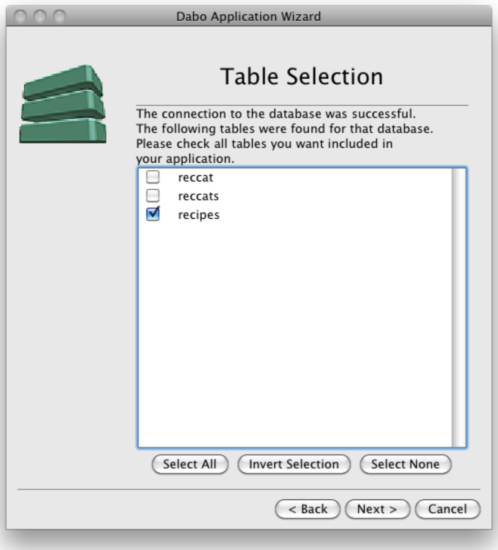
Press 'Next' to enter database parameters.



Enter the parameters here, and then click 'Next'.

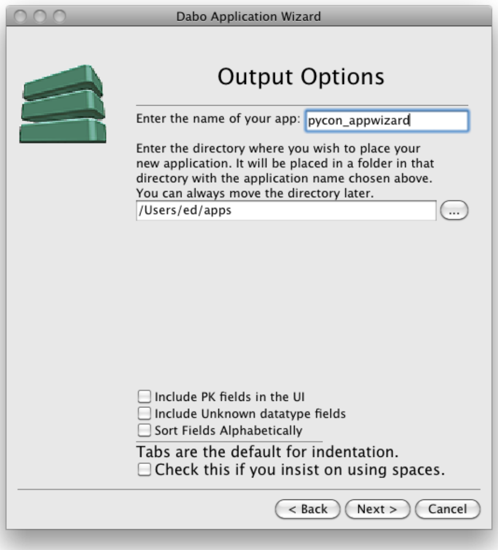
Profile:	MySQL
DbType:	MySQL
Name:	pycon
Host:	dabodev.com
Database:	pycon
User:	pycon
Password:	•••••••
Port:	3306

At this point, the wizard will connect to the database and get the relevant table information. If you mis-typed anything above, or if the server is not available, you will get an error message explaining what went wrong. Assuming that all went well, you will see the page shown below left, containing the available tables in the database. For this example, let's just check the 'recipes' table, and then click 'Next'. That will bring you to the page below right.



The connection to the database was successful.
The following tables were found for that database.
Please check all tables you want included in your application.

- [ ] reccat
- [ ] reccats
- [x] recipes



Enter the name of your app: pycon_appwizard

Enter the directory where you wish to place your new application. It will be placed in a folder in that directory with the application name chosen above. You can always move the directory later.

/Users/ed/apps

- [ ] Include PK fields in the UI
- [ ] Include Unknown datatype fields
- [ ] Sort Fields Alphabetically

Tabs are the default for indentation.
- [ ] Check this if you insist on using spaces.

It will default the name of your app to the database name; let's make it more specific, and change the app name to 'pycon_appwizard'. You can also select the directory where you want Dabo to place your app.

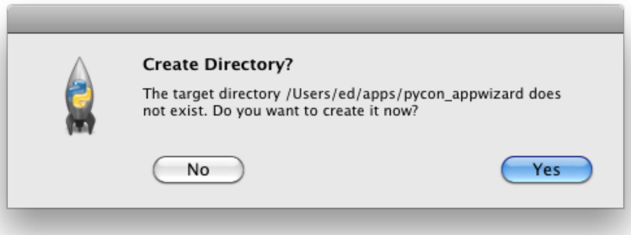
There are options at the bottom that let you control aspects of the generated app. First, you generally don't want the PK columns in your tables as part of your UI; users shouldn't care about PKs. However, some older systems used PKs whose values are significant, so if you are dealing with a table like that, check the first box at the bottom.

The next checkbox controls how Dabo deals with 'exotic' data types that are usually specific to a particular RDBMS. If your data contains such columns, and you want Dabo to try to convert them automatically, check the second box.

The next checkbox controls how the fields are added to the UI. Some people use a naming convention that orders fields alphabetically in a significant way; if this is important to you, check this box. Otherwise, Dabo adds the fields to the UI in the order they are reported from the database. You can always re-order them for display in the UI later.

Finally, we mentioned earlier that Dabo uses tabs for indentation by default. If this is pure heresy to you, check this last box, and Dabo will dutifully use 4 spaces per indentation level.

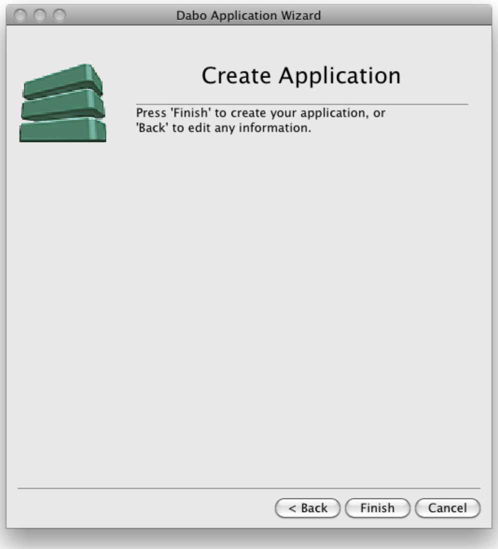
Now click 'Next', and you may see the confirmation dialog:



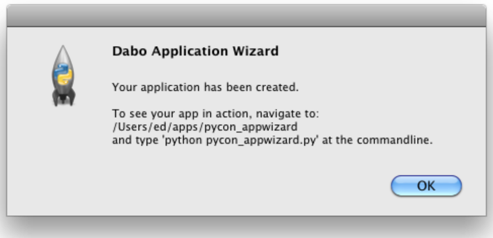
Create Directory?

The target directory /Users/ed/apps/pycon_appwizard does not exist. Do you want to create it now?

Assuming that creating this directory is what you want, click 'Yes'. Otherwise, click 'No' and you will be returned to the previous screen, where you can pick a different location. Once you have the location set, there is one last screen, shown below left. This is the final confirmation; if you click 'Finish', the app is created. If you need to change anything, you can click 'Back' and alter any of the values you selected earlier. Or just click 'Cancel' if you want to bail. Assuming that you clicked 'Finish', you'll see the notice below right.



Press 'Finish' to create your application, or 'Back' to edit any information.

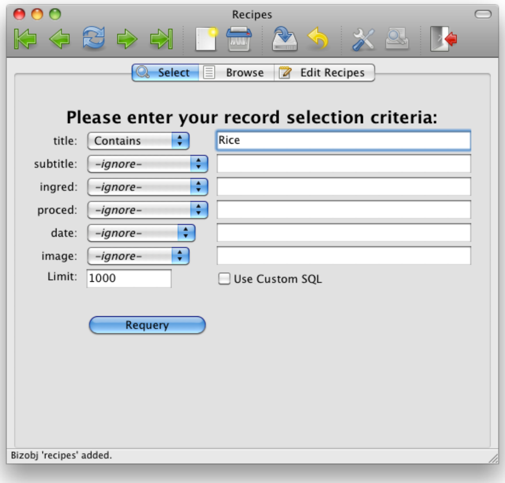


Dabo Application Wizard

Your application has been created.

To see your app in action, navigate to:
/Users/ed/apps/pycon_appwizard
and type 'python pycon_appwizard.py' at the commandline.

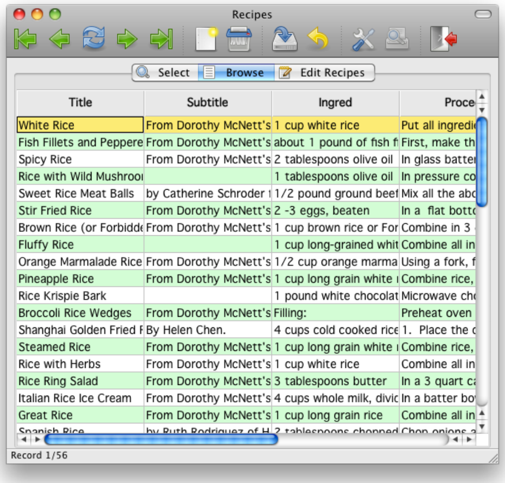
That's it! You can now navigate to the directory you chose for your app, and run it! Here's what it looks like (lower left). Note that it created search selectors for each of the fields. For this example, I want to see all the recipes that contain the word 'Rice' in the title, so I enter these values and click the 'Requery' button. I then get a grid where I can browse all the matches. Note that the status bar informs me that there are 56 matched records.



Please enter your record selection criteria:

title:	Contains	Rice
subtitle:	-ignore-	
ingred:	-ignore-	
proced:	-ignore-	
date:	-ignore-	
image:	-ignore-	
Limit:	1000	Use Custom SQL

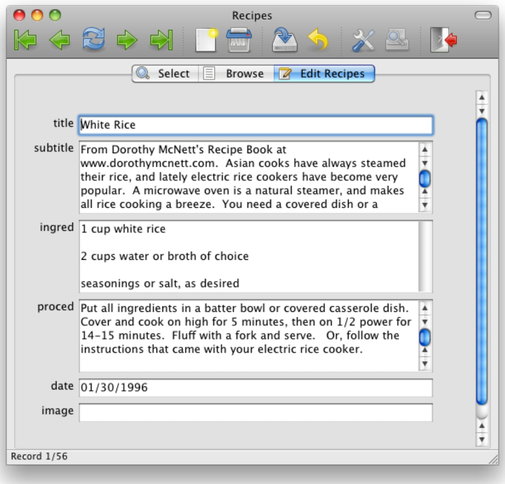
Bizobj 'recipes' added.



Title	Subtitle	Ingred	Proce
White Rice	From Dorothy McNett's	1 cup white rice	Put all ingredi
Fish Fillets and Peppere	From Dorothy McNett's	about 1 pound of fish fi	First, make th
Spicy Rice	From Dorothy McNett's	2 tablespoons olive oil	In glass batter
Rice with Wild Mushroo		1 tablespoons olive oil	In pressure co
Sweet Rice Meat Balls	by Catherine Schroder t	1/2 pound ground beef	Mix all the abo
Stir Fried Rice	From Dorothy McNett's	2 -3 eggs, beaten	In a flat botto
Brown Rice (or Forbidde	From Dorothy McNett's	1 cup brown rice or For	Combine in 3
Fluffy Rice		1 cup long-grained whit	Combine all in
Orange Marmalade Rice	From Dorothy McNett's	1/2 cup orange marma	Using a fork, f
Pineapple Rice	From Dorothy McNett's	1 cup long grain white	Combine rice,
Rice Krispie Bark		1 pound white chocolat	Microwave ch
Broccoli Rice Wedges	From Dorothy McNett's	Filling:	Preheat oven
Shanghai Golden Fried F	By Helen Chen.	4 cups cold cooked rice	1. Place the c
Steamed Rice	From Dorothy McNett's	1 cup long grain white	Combine rice,
Rice with Herbs	From Dorothy McNett's	1 cup white rice	Combine all in
Rice Ring Salad	From Dorothy McNett's	3 tablespoons butter	In a 3 quart ca
Italian Rice Ice Cream	From Dorothy McNett's	4 cups whole milk, divic	In a batter bo
Great Rice	From Dorothy McNett's	1 cup long grain rice	Combine all in
Spanish Rice	by Ruth Rodriguez of H	2 tablespoons chopped	Chop onions a

Record 1/56

From here I can select the record I want to edit, and either click the 'Edit Recipes' tab, or simply press 'Enter'. I then get the editing screen:



title White Rice

subtitle From Dorothy McNett's Recipe Book at www.dorothymcnett.com. Asian cooks have always steamed their rice, and lately electric rice cookers have become very popular. A microwave oven is a natural steamer, and makes all rice cooking a breeze. You need a covered dish or a

ingred 1 cup white rice

2 cups water or broth of choice

seasonings or salt, as desired

proced Put all ingredients in a batter bowl or covered casserole dish. Cover and cook on high for 5 minutes, then on 1/2 power for 14-15 minutes. Fluff with a fork and serve. Or, follow the instructions that came with your electric rice cooker.

date 01/30/1996

image

Record 1/56

From this screen I can change any of the values and then save them back to the database, or I can navigate to another record, or delete the record, or just about anything else that you would need to do in a simple table maintenance app.

So let's step back for a minute and think about what we just accomplished: we have a complete table maintenance app that allows us to search the database, select from the results, and then edit and save our changes back. We have a complete UI for this that was constructed for us using information gathered from the database. And we did this in under a minute!

Customizing an AppWizard Application

AppWizard generates a full-blown application with a menu system, forms, reports, bizobjs, and a file to support your default connection to the database. You even get a setup.py and supporting files to build your app for distribution to Mac, Windows, and Linux in native executable format.

You can stop there and just accept what was automatically generated — perhaps the generated output is just fine for your simple needs — or you can customize the code to your liking. The scope of your enhancements really isn't limited: over the past 2 years, I (Paul) have developed a cross-platform commercial application with twenty modules starting off with the output of the AppWizard against one table. It served as a starting point for a quick prototype, and there was no apparent reason to start from scratch when time to start building the real application. Treat the code generated using the AppWizard as a starting point; going forward from there, you can use any part of Dabo that you desire.

For the enhancements in this tutorial, we'll concentrate first on some "low-hanging fruit" in the generated AppWizard code, and then move on to some more advanced topics. Specifically, we'll alter the captions of fields on the grid and in the select and edit pages, add some business rules, add some UI controls to link recipe categories to recipes, and make a report for printing out recipes. We'll do this in under 30 minutes.

Preliminaries

First, run AppWizard again, but this time include all 3 recipe-related tables (**recipes**, **reccats**, and **reccat**). The tables are:

- recipes : table of recipe names, ingredients, and procedures.
- reccats : table of recipe categories (Main Dishes, etc.)
- reccat : intermediary 'join' table for a many:many relationship between recipes and reccats.

Also, to be consistent with my examples that follow, save the app as 'recipes_aw' instead of 'pycon_appwizard' like we did the first time through.

Second, remove **reccat** from most of the UI. This is a join table, so we won't be interacting with it directly. AppWizard didn't know this and dutifully included it. Open **ui/MenFileOpen.py** and simply make the forms definition on **line 24** look like this:

```
24     forms = (
25         ("&Recipes", app.ui.FrmRecipes),
26         ("&Categories", app.ui.FrmReccats))
```

Note that I removed the 'reccats' entry, spelled out 'Categories', swapped the order of the Recipes and Categories entries, and added explicit hot keys (&). I also removed an extraneous spacer.

The **MenFileOpen** object is instantiated and added to the **File|Open** menu when instantiating any of the forms in our application. You can see this calling code in, for example, **ui/FrmRecipes.py**.

When the AppWizard generated the application, it unfortunately picked 'reccat' as the default form to open when running the application. But fortunately, this gives us an opportunity to dig around in the main script. Change the default in **line 50** of the main file (**recipes_aw.py** in the root directory):

```
50 default_form = ui.FrmRecipes
```

Once you save your changes and run the application, the Recipes form will open by default.

The main file is the entry point of the application. It imports dabo, and instantiates our application's App class, which is a subclass of **dabo.dApp()**. Recall that Dabo has 3-tiers (db, biz, ui). The Application object encompasses all these tiers, so no matter what tier you are coding in, you can always get at the dApp instance. The main file that the AppWizard generated makes some convenient links for you, such as adding the ui and biz namespaces to the dApp instance, and saving the automatically-opened db connection to the attribute 'dbConnection'. So, to instantiate a bizobj, you wouldn't need to import anything, you'd just need something like:

```
app = self.Application
mybiz = app.biz.Recipes(app.dbConnection)
```

The code at the bottom of the main script is what bootstraps your application. It is what gets executed first when you '**python recipes_aw.py**'.

Captions

The captions of the various UI elements are the easiest to change, and the activity of changing the captions serves as a jumping-off point for discovering how the code is organized. Use a text searching utility like grep to find the caption you want to change. Once you've changed that, look around in the file you've opened to change other things to your liking as well.
For example, you probably want to find and replace 'ingred' with 'ingredients'. First, find the files in the UI directory that contain this string, and then open vim (or your favorite text editor) to make the changes.

```
mac:ui pmcnett$ grep -i ingred *.py
GrdRecipes.py: self.addColumn(dabo.ui.dColumn(self,
DataField="ingred", Caption="Ingred",
PagEditRecipes.py: ## Field recipes.ingred
PagEditRecipes.py: label = self.addObject(dabo.ui.dLabel,
NameBase="lblingred",
PagEditRecipes.py: Caption="ingred")
PagEditRecipes.py: objectRef = self.addObject(dabo.ui.dEditBox,
NameBase="ingred",
PagEditRecipes.py: DataSource="recipes", DataField="ingred")
PagSelectRecipes.py: ## Field recipes.ingred
PagSelectRecipes.py: lbl.Caption = "ingred:"
PagSelectRecipes.py: lbl.relatedDataField = "ingred"
PagSelectRecipes.py: self.selectFields["ingred"] = {
PagSelectRecipes.py: dabo.errorLog.write("No control class found for
field 'ingred'.")
```

We quickly find that 'ingred' (either upper or lower case) is found in **GrdRecipes.py**, **PagEditRecipes.py**, and **PagSelectRecipes.py**. These represent the classes for the browse grid, the edit page, and the select page, respectively. So open up **GrdRecipes.py** and make the change, taking time to also look at the rest of the file to understand what it is doing. First, there's a line that defines the encoding to use. This is standard Python. Then we import dabo, and set the UI toolkit to wxPython. Then we import the superclass grid definition from GrdBase.py, which in turn is a subclass of **dabo.lib.datanav.Grid**. Finally, we define our **GrdRecipes** class based on **GrdBase**.

```
# -*- coding: utf-8 -*-

import dabo
if __name__ == "__main__":
    dabo.ui.loadUI("wx")
from GrdBase import GrdBase

```

```python

class GrdRecipes(GrdBase):
```

The rest of the file overrides the default **afterInit()** method, and adds the columns to the grid that you see in the browse page when you run the recipes application. One of these definitions looks like:

```python
    self.addColumn(dabo.ui.dColumn(self, DataField="ingred",
            Caption="Ingred", Sortable=True, Searchable=True,
            Editable=False))
```

And, while we initially set out to change the caption from 'Ingred' to 'Ingredients', this field actually doesn't make that much sense to include in the browse grid at all, since the intent of the grid is to pick records to display in the edit page, and while it is possible to configure the grid to show multiple lines of text for each field, doing that would make it less effective as a record picker. The ingredients are not critical information to display in this grid, so the column should be removed. Remove that **addColumn()** call completely, as well as those for **subtitle**, **proced**, and **image**. For fun, add some custom property settings for the remaining columns, such as **HeaderBackColor**, **ForeColor**, **HeaderFontItalic**, and **FontSize**. Here's my complete **afterInit()** after making some of these changes:

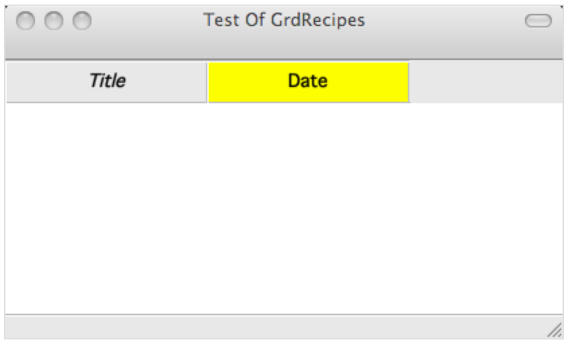
```python
    def afterInit(self):
        self.super()

        self.addColumn(dabo.ui.dColumn(self, DataField="title",
                Caption="Title", HeaderFontItalic=True,
                FontBold=True)
        self.addColumn(dabo.ui.dColumn(self, DataField="date",
                Caption="Date", HeaderBackColor="yellow",
                FontSize=12))
```

Note that I removed some of the generated property settings (**Sortable**, **Searchable**, and **Editable**). These aren't needed because they already default to the specified values, and code should be as clean and concise as possible. They were included in the generated code to help you get started, for the same reason that the files are heavily commented.

There's some test code at the bottom of GrdRecipes.py:

```python
if __name__ == "__main__":
    from FrmRecipes import FrmRecipes
    app = dabo.dApp(MainFormClass=None)
    app.setup()
    class TestForm(FrmRecipes):
        def afterInit(self): pass
    frm = TestForm(Caption="Test Of GrdRecipes", Testing=True)
    test = frm.addObject(GrdRecipes)
    frm.Sizer.append1x(test)
    frm.show()
    app.start()
```

Putting test code such as this in your python scripts allows you to quickly run a test of your layout without running the entire app, speeding up the development cycle. In this case, since there's no data, you'll only see the property changes of the header, but go ahead and run 'python GrdRecipes.py' to see what it looks like. You should see something like:



Okay, so we got sidetracked from the initial search of 'ingred' but we ended up with an edited **GrdRecipes.py** file that is likely what we want. Why not complete the thought and edit the other **Grd*.py** files while we are at it?

```
mac:ui pmcnett$ ll Grd*.py
-rw-r--r-- 1 pmcnett pmcnett 97  Feb 8 11:19 GrdBase.py
-rw-r--r-- 1 pmcnett pmcnett 808 Feb 8 11:19 GrdReccat.py
-rw-r--r-- 1 pmcnett pmcnett 944 Feb 8 11:19 GrdReccats.py
-rw-r--r-- 1 pmcnett pmcnett 735 Feb 8 11:39 GrdRecipes.py
```

GrdBase.py is just a simple subclass of **dabo.lib.datanav.Grid** which the other grid classes inherit from. If you want common look, feel, or functionality in all your grids, you'd put that into **GrdBase** instead of duplicating it in all grid classes.

We aren't going to be using **GrdReccat.py**, as there's no immediate reason to browse this intermediary join table directly. However, it doesn't hurt to leave it in place, untouched, in case it becomes convenient to use in the future. For now, move on and edit **GrdReccats.py** to remove the extraneous columns and to change the caption 'Descrp' to 'Title', which better describes the field to the user. Here's the entire class definition after those changes:

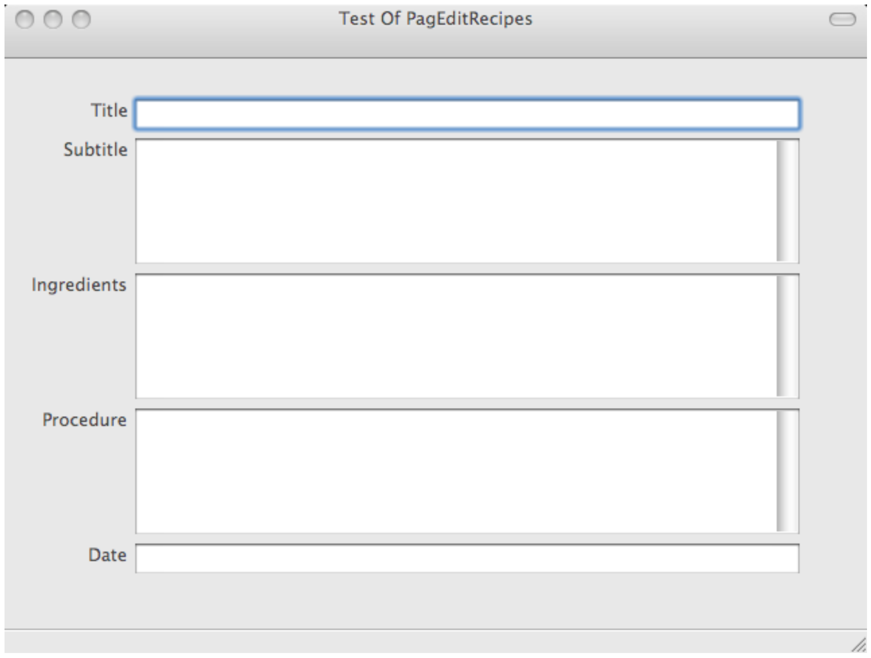
```
class GrdReccats(GrdBase):

     def afterInit(self):
         self.super()
         self.addColumn(dabo.ui.dColumn(self, DataField="descrp",
                 Caption="Title"))
```

Now that the grids are shaped up how we want them, let's continue on our Caption-changing adventure with **PagEditRecipes.py**. This file defines the page class for displaying and editing your fields in the current row of the recipes bizobj. Notice when you run the app, the basic UI is a pageframe with 3 pages: Select, Browse, and Edit. You use these pages to enter your selection criteria and requery the bizobj, to browse or navigate the records in the bizobj, and to edit the current record, respectively. **PagEditRecipes** is the edit page for the recipes module.

Open up **PagEditRecipes** in your text editor and you get to see the code that lays out the various **dLabel**s, **dTextBox**es, and **dEditBox**es on the page. You can see the code to create the **dGridSizer** which is used for automatic control positioning so that you don't need to predict how to handle screens of differing sizes or cross-platform issues. Search for 'ingred' and find **line 39**:

```
39      ## Field recipes.ingred
40      label = self.addObject(dabo.ui.dLabel, NameBase="lblingred",
41              Caption="ingred")
42      objectRef = self.addObject(dabo.ui.dEditBox, NameBase="ingred",
43              DataSource="recipes", DataField="ingred")
```

It's the label definition that contains the caption we want to change. In addition to changing that label's caption to 'Ingredients', go to the other objects as well and make their captions have title case. Run the test to see how it looks by typing '**python PagEditRecipes**' at the terminal.



The automatic layout, while acceptable, can be tweaked easily to make it better match the relative importance of the data. While even more control could be gained by switching from the generated **dGridSizer** to nested **dSizer**s, we can gain some desired changes quickly and easily by making some simple tweaks:

Make the recipe title stand out more by making the font bold and bigger (**line 22**):

```
17      ## Field recipes.title
18      label = self.addObject(dabo.ui.dLabel, NameBase="lbltitle",
19              Caption="Title")
20      objectRef = self.addObject(dabo.ui.dTextBox, NameBase="title",
21              DataSource="recipes", DataField="title",
22              FontBold=True, FontSize=12)
```

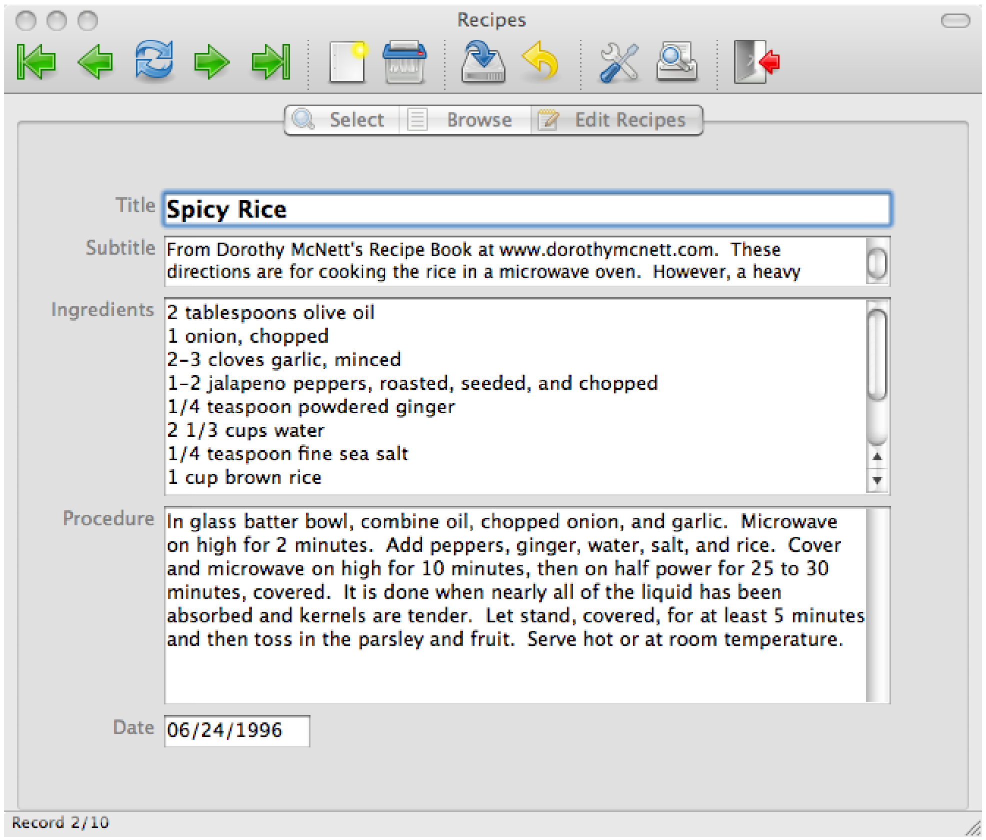
Make the subtitle a smaller font size, and keep it fixed at a smaller height rather than growing and shrinking along with the other fields when the form is resized (I know that this field just doesn't need that much space in practice) (**line 33** sets the minimum height while **line 37** tells the grid sizer not to grow that particular row):

```
28      ## Field recipes.subtitle
29      label = self.addObject(dabo.ui.dLabel, NameBase="lblsubtitle",
30              Caption="Subtitle")
31      objectRef = self.addObject(dabo.ui.dEditBox,
32              NameBase="subtitle", DataSource="recipes",
33              DataField="subtitle", Height=35, FontSize=9)
34
35      gs.append(label, alignment=("top", "right") )
36      currRow = gs.findFirstEmptyCell()[0]
37      gs.setRowExpand(False, currRow)
```

The last thing to address is the **recipe date** field, which is way too wide to display a date in, and we should really be using a **dabo.ui.dDateTextBox** instead of a raw **dTextBox**, since the former adds some nice features such as a drop-down calendar widget (**line 68** changes the class we are instantiating; **line 72** removes the 'expand' keyword which will keep the control at its default Width (override the default if you like by specifying the **Width** property):

```
65      ## Field recipes.date
66      label = self.addObject(dabo.ui.dLabel, NameBase="lbldate",
67              Caption="Date")
68      objectRef = self.addObject(dabo.ui.dDateTextBox,
69              NameBase="date", DataSource="recipes",
70              DataField="data")
71      gs.append(label, alignment=("top", "right") )
72      gs.append(objectRef)
```

Here's the edit page for recipes now, with some live data. Much better with a little tweaking!



Title: Spicy Rice

Subtitle: From Dorothy McNett's Recipe Book at www.dorothymcnett.com. These directions are for cooking the rice in a microwave oven. However, a heavy

Ingredients:
2 tablespoons olive oil
1 onion, chopped
2-3 cloves garlic, minced
1-2 jalapeno peppers, roasted, seeded, and chopped
1/4 teaspoon powdered ginger
2 1/3 cups water
1/4 teaspoon fine sea salt
1 cup brown rice

Procedure: In glass batter bowl, combine oil, chopped onion, and garlic. Microwave on high for 2 minutes. Add peppers, ginger, water, salt, and rice. Cover and microwave on high for 10 minutes, then on half power for 25 to 30 minutes, covered. It is done when nearly all of the liquid has been absorbed and kernels are tender. Let stand, covered, for at least 5 minutes and then toss in the parsley and fruit. Serve hot or at room temperature.

Date: 06/24/1996

Record 2/10

May as well take a minute to change **PagEditReccats.py** to our liking, too. Change the caption of 'descrp' to 'Title' and 'html' to 'Description'. The field names in this case don't reflect the meaningful information, and it is of utmost importance to not confuse the user.

Now for the select pages: The generated code in there is repetitive and obtuse. We'll come back to it later — for now just change the captions like 'ingred' to 'Ingredients', etc. Search on 'Caption' to find them. Do this for both **PagSelectRecipes.py** and **PagSelectReccats.py**.

Business Rules

At the moment, you are allowed to make any changes you desire to the recipes using the UI, including adding blank new records, deleting existing records, and the like. By default, the bizobjs allow everything all the time, so let's add some restrictions on the scope of changes a normal user is able to make. Dabo is designed to allow fairly fine-grained control over all database operations, so you'd be able to make an app, for instance, that allowed a certain class of users - say, "power users" - more access than normal "users". You would write the code for this with the help of the Application's SecurityProvider and login screen, which is out of scope for this tutorial. For the purposes of this example, we'll make some simple business rules that apply to everyone using the application, and show them in operation.

Follow along as I edit **biz/Recipes.py**.

One of the business rules is that all records must have a valid date. Write your business objects to be as nice to the user as possible, which includes setting sensible and valid defaults. In the case of the date field, a good starting value is today's date. Set this using the DefaultValues property of the Recipes bizobj (**lines 3 and 25**):

```
1 # -*- coding: utf-8 -*-
2
3 import datetime
4 from Base import Base
5
6
7 class Recipes(Base):
8
9   def initProperties(self):
10     self.super()
11     self.Caption = "Recipes"
12     self.DataSource = "recipes"
13     self.KeyField = "id"
14
15     self.DataStructure = (
16         ("id", "I", True, "recipes", "id"),
17         ("title", "C", False, "recipes", "title"),
18         ("subtitle", "M", False, "recipes", "subtitle"),
19         ("ingred", "M", False, "recipes", "ingred"),
20         ("proced", "M", False, "recipes", "proced"),
21         ("date", "D", False, "recipes", "date"),
22         ("image", "C", False, "recipes", "image"),
23     )
24
25     self.DefaultValues["date"] = datetime.date.today
```

Note that we set the default value of the date field to the datetime.date.today function object, not the return value of calling the function. Dabo will get the return value of the function at the time any new records are added by using this style. In this case it doesn't matter much, because who's going to run the application over a day, but this feature is good to remember. I use it in some of my applications to fetch the next order number, customer number, etc.

Another sensible rule is to not allow empty, or near-empty, recipe titles. Below, we try a couple different validation methods, finally settling on validateRecord() instead of validateField(), because we want to give as much control over the flow of the data-entry experience to the user as possible. We only really care that the data adheres to the rules at the time of save(), so validateRecord() is really the way to go. But go ahead and comment **line 29** to see **validateField()** in action, as well.

```
28   def validateField(self, fld, val):
29     return ""  ## validateRecord is better
30     if fld == "title":
31       return self.validateTitle(val)
32
33
34   def validateRecord(self):
```

```
35     msgs = []
36     msg = self.validateTitle()
37     if msg:
38       msgs.append(msg)
39     if not self.Record.date or self.Record.date < datetime.date(1990, 1, 1):
40       msgs.append("There must be a valid date.")
41
42     if msgs:
43       return "The recipe '%s' could not be saved because:\n" \
44           % self.Record.title.strip() + "\n".join(msgs)
45     return ""  ## all okay
46
47
48   def validateTitle(self, val=None):
49     oldVal = self.oldVal("title")  ## as it was after requery()
50     curVal = self.Record.title     ## as it is now
51     if val is None:
52       # val is only sent in validateField, and is the user-entered
value
53       # not yet in the bizobj at all.
54       val = curVal
55     if oldVal.strip() and len(val.strip()) < 2:
56       return "Title too short.."
```

Above, we overrode the **validateField()** and **validateRecord()** methods of all dBizobj-derived classes, and we added our own method, **validateTitle()**, which is called from both the other methods. Any return value gets shown in the UI in some fashion, but here at the bizobj tier we don't care what the UI does with the information: we only care to write the logic to determine if the business rules pass or fail, and if they fail, to return a meaningful message.

One of the rules we set was that the date must be greater than January 1, 1990. Go ahead and run the app, add a new record (using the navigation toolbar, the File|New menu item, or by pressing Ctrl+N (Cmd+N on Mac)), and try to save it. You should get a "Save Failed" dialog with the message that the recipe title is too short. The date field passed validation because of the default we set in **line 25**. Okay, now enter a meaningful recipe title, but change the date to something prior to 1990. Again, it won't let you save. Only when all rules set in validateRecord() pass, will the save be allowed to continue.

Had this bizobj been a child of another bizobj, and the validateRecord() was being called as a result of the parent bizobj's save() method, the failure would have resulted in the rolling back of all changes upstream. All bizobj methods involved in changing data (updates and inserts) are wrapped in transactions (a single transaction for each save, which would encompass the resulting saves of child bizobjs down the line).

Editing these rules gave you the opportunity to take a look at the Recipe bizobj class definition. Note that we define a DataStructure explicitly: doing this keeps Dabo from guessing field types for you - it puts you in control since you know best. Likely, the defaults generated by the AppWizard are just fine (they are what Dabo would have guessed at runtime), but in case one of them is reflecting the wrong type, you can change it here.

Dabo uses one-character short type identifiers in this context, and you can see how they map to python types by taking a look at **dabo.db.getPythonType()** and **dabo.db.getDaboType()**. Each of these rely on a mapping, which you can also print to understand what's going on behind the scenes:

```
mac:biz pmcnett$ python
Python 2.5.4 (r254:67917, Dec 23 2008, 14:57:27)
[GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import dabo
>>> print dabo.db.getPythonType("C")
<type 'unicode'>
>>> print dabo.db.getDaboType(bool)
B
>>> print dabo.db.daboTypes
{'C': <type 'unicode'>, 'B': <type 'bool'>, 'D': <type
'datetime.date'>, 'G': <type 'long'>, 'F': <type 'float'>, 'I': <type
'int'>, 'M': <type 'unicode'>, 'L': <type 'str'>, 'N': <class
'decimal.Decimal'>, 'T': <type 'datetime.datetime'>}
```

So you can find the codes to enter in the DataStructure by inspecting **dabo.db.daboTypes**.

Creating the Many:Many Relationship between Recipes and Categories

Currently, we can edit, add, and delete recipes. We can edit, add, and delete categories. But we can't see what categories are linked to a given recipe, nor can we see what recipes are linked to a given category. And perhaps most importantly, we can't make changes to what recipes are in what category, and vice-versa.

Adding the desired functionality requires setting up relationships between the relevant bizobjs, and adding UI controls to interact with the child bizobjs. Because the bizobj relationships need to be in place before anything will work at the ui layer, we'll do that first.

Recall that there are currently 3 bizobjs defined: Recipes, Reccat, and Reccats. Reccat is the many:many 'joiner' table, that contains foreign keys to both Recipes and Reccats. To give the UI the ability to show what categories are linked to each recipe, we need to create a suitable child bizobj for Recipes. Since it is easy, and we may want the functionality in the future, we'll do a similar thing the other way around to the Reccats bizobj as well, so we can view recipes linked to a certain category.

In the listing below of biz/Reccat.py, I've removed the lengthy comments, added some whitespace, and added two subclasses of Reccat: CategoriesForRecipe and RecipesForCategory. Take some time to look over this code as there are some subtle changes.

```
 # -*- coding: utf-8 -*-

from Base import Base
```

```
class Reccat(Base):
    def initProperties(self):
        self.super()
        self.Caption = "Reccat"
        self.DataSource = "reccat"
        self.KeyField = "id"

        self.DataStructure = [
                ("id", "I", True, "reccat", "id"),
                ("recid", "I", False, "reccat", "recid"),
                ("catid", "I", False, "reccat", "catid"),
        ]

        self.FillLinkFromParent = True


    def afterInit(self):
        self.super()


    def setBaseSQL(self):
        self.addFrom("reccat")
        self.setLimitClause("500")
        self.addFieldsFromDataStructure()



class CategoriesForRecipe(Reccat):
    def initProperties(self):
        self.LinkField = "recid"
        self.super()
        self.DataStructure.append(("catname", "C", False, "reccats", "descrp"))


    def setBaseSQL(self):
        self.super()
        self.addJoin("reccats", "reccats.id = reccat.catid", "inner")



class RecipesForCategory(Reccat):
    def initProperties(self):
        self.LinkField = "catid"
        self.super()
        self.DataStructure.append(("recname", "C", False, "recipes", "title"))


    def setBaseSQL(self):
        self.super()
        self.addJoin("recipes", "recipes.id = reccat.recid", "inner")
```

First, I've changed the DataStructure from a tuple, which is read-only, to a list, so that the subclasses can augment it. Second, I've set the FillLinkFromParent property to True, so we don't need to worry about explicitly adding the foreign key value when adding new records. Third, in the subclasses I've set the LinkField property to tell Dabo which is the foreign key to the parent, so that when the parent requery()s, the child can requery along with it automatically. Fourth, I've added inner joins to get the recipe name from the categories bizobj, and the category name from the recipes bizobj, respectively. I used subclassing for maximized code-reuse.

Next, there are some changes to the Recipes and Reccats bizobjs to add the appropriate child to each. I'll just show the changes to Recipes here, since that's what we are focused on and the changes are similar. First (line 5), add the import so Python can find the class definition:

```
1 # -*- coding: utf-8 -*-
2
3 import datetime
4 from Base import Base
5 from Reccat import CategoriesForRecipe
```

Then (lines 27-33, starting at the end of the initProperties() method), add the code to add the child. I coded it this way in case future enhancements result in more than one child being added.

```
26     self.DefaultValues["date"] = datetime.date.today
27     self.addChildren()
28
29
30   def addChildren(self):
31     app = self.Application
32     self.bizReccat = CategoriesForRecipe(app.dbConnection)
33     self.addChild(self.bizReccat)
```

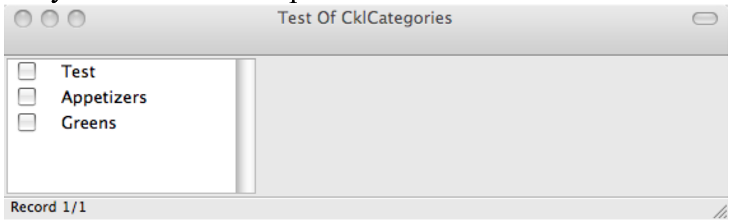
Now the Recipes bizobj has a child that can contain zero to many records representing the categories assigned to each recipe. Everything is in place for the UI to be able to display this information. Next, we add a couple methods to FrmRecipes, including instantiating a Reccats bizobj for the purpose of maintaining a list of all the possible categories.

```
24   def afterInit(self):
25     if not self.Testing:
26       app = self.Application
27       self.addBizobj(app.biz.Recipes(app.dbConnection))
28
29       # Add the separate categories bizobj, to give us the list of
all
30       # recipe categories:
31       self.addBizobj(app.biz.Reccats(app.dbConnection))
32       self.requeryCategories()
33     self.super()
34
35
36   def requeryCategories(self):
37     biz = self.getBizobj("reccats")
38     biz.UserSQL = "select * from reccats order by descrp"
39     biz.requery()
40
41
42   def getCategoryChoicesAndKeys(self):
    """Return two lists, one for all descrp values and one for all id values."""
    choices, keys = [], []
    biz = self.getBizobj("reccats")
    for rownum in biz.bizIterator():
      choices.append(biz.Record.descrp)
      keys.append(biz.Record.id)
    return (choices, keys)
```

These are form-level methods because they can be potentially called upon by any control in the form. In general, keep code that interacts with bizobjs at the form level - it is just easier to maintain in the long run. Next, we create a new file in the UI directory, CklCategories.py, which defines a dCheckList subclass, which will allow the UI to show a listing of all possible categories, while putting a checkmark next to all the categories linked up with the current recipe:

```
# -*- coding: utf-8 -*-

import dabo
if __name__ == "__main__":
    dabo.ui.loadUI("wx")



class CklCategories(dabo.ui.dCheckList):
    def initProperties(self):
        self.Width = 200
        self._needChoiceUpdate = True


    def updateChoices(self):
        self._needChoiceUpdate = False
        if self.Form.Testing:
            self.Choices = ("Test", "Appetizers", "Greens")
            self.Keys = (1, 2, 3)
            return
        self.Choices, self.Keys = self.Form.getCategoryChoicesAndKeys()


    def update(self):
        self.super()

        if self._needChoiceUpdate:
            self.updateChoices()

        if self.Form.Testing:
            return

        bizRecipes = self.Form.getBizobj()
        bizReccat = bizRecipes.bizReccat
        keyvalues = []
        for rownum in bizReccat.bizIterator():
            keyvalues.append(bizReccat.Record.catid)
        self.KeyValue = keyvalues
    def onHit(self, evt):
        print "onHit", self.KeyValue


if __name__ == "__main__":
    from FrmRecipes import FrmRecipes
    app = dabo.dApp(MainFormClass=None)
    app.setup()
    class TestForm(FrmRecipes):
        def afterInit(self): pass
    frm = TestForm(Caption="Test Of CklCategories", Testing=True)
    test = frm.addObject(CklCategories)
    frm.Sizer.append(test, 1)
    frm.show()
    frm.update()
    app.start()
```

The Testing property allows us to code our classes in such a way as to allow testing the control without having to load the entire app, or real data. Go ahead and run this file once you've copied and pasted the code, and you should see output like:



When you put a checkmark on an item, the onHit() event handler fires, and you can see the print output on the console. We'll come back and fill in the onHit() with real functionality later - when done, it will either add a new record to the child bizobj, or delete an existing record from the child bizobj.

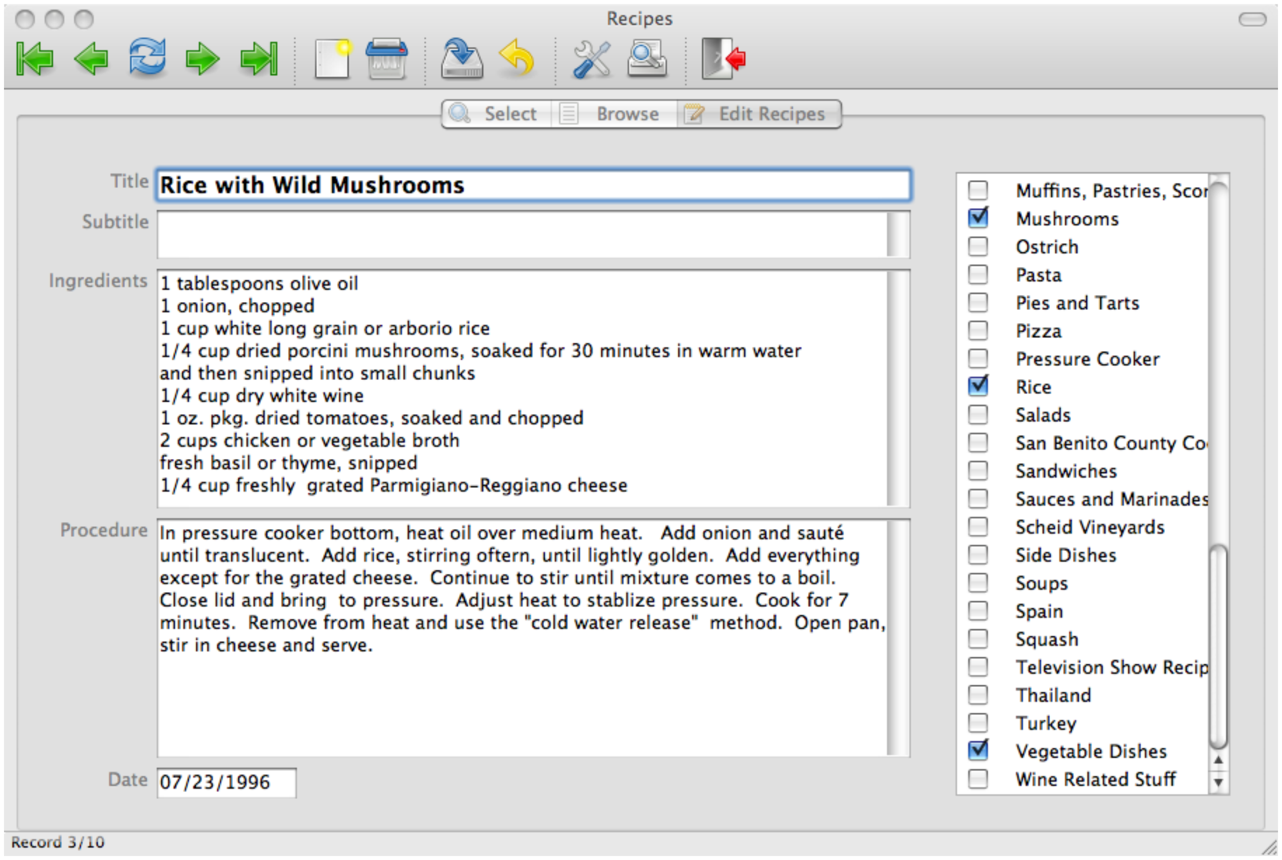
Now it's time to add this new CklCategories control to the edit page of the recipes form. So, edit ui/PagEditRecipes.py, and add the import on line 7, and change the main sizer from a vertical orientation to horizontal (line 16), so that when we add the new control it goes to the right of the grid sizer, instead of below it:

```
 6 from PagEditBase import PagEditBase
 7 from CklCategories import CklCategories
 8
 9
10
11 class PagEditRecipes(PagEditBase):
12
13   def createItems(self):
14     """Called by the datanav framework, when it is time to create
the controls."""
15
16     mainSizer = self.Sizer = dabo.ui.dSizer("h")
```

Next, we change the insertion of the dGridSizer (gs) to remove the border from the right (line 79), we instantiate and append to the main sizer a new vertical dSizer (lines 81 and 82), with no left border, and finally we add the new CklCategories control to the new sizer. Figuring out the sizer properties was a result of editing, running PagEditRecipes.py to check the layout, and editing again. Getting the expand and border settings right never happens the first time for me, after over five years of dealing with sizers. Luckily it is easy and quick to test-edit-repeat until the design looks good. Later on, you'll see how to use the Class Designer to interactively see what various property settings have on sizer layout.

```
79      mainSizer.insert(0, gs, 1, "expand", border=20, borderSides=("top", "bottom", "left'
80
81      vsRight = dabo.ui.dSizer("v")
82      mainSizer.append(vsRight, "expand", border=20, borderSides=("top", "bottom", "right'
83
84      vsRight.append(CklCategories(self), 1, "expand")
85
86      self.Sizer.layout()
87      self.itemsCreated = True
88
89      self.super()
```

With these changes, we can run the application, select some search criteria, requery, and when we get to the edit page we should see some checked categories for each recipe. As we navigate through the list of recipes we should see the checked categories change. Here's a screenshot from my system at this time:



Title: Rice with Wild Mushrooms

Subtitle:

Ingredients:
1 tablespoons olive oil
1 onion, chopped
1 cup white long grain or arborio rice
1/4 cup dried porcini mushrooms, soaked for 30 minutes in warm water
and then snipped into small chunks
1/4 cup dry white wine
1 oz. pkg. dried tomatoes, soaked and chopped
2 cups chicken or vegetable broth
fresh basil or thyme, snipped
1/4 cup freshly grated Parmigiano-Reggiano cheese

Procedure:
In pressure cooker bottom, heat oil over medium heat. Add onion and sauté
until translucent. Add rice, stirring oftern, until lightly golden. Add everything
except for the grated cheese. Continue to stir until mixture comes to a boil.
Close lid and bring to pressure. Adjust heat to stablize pressure. Cook for 7
minutes. Remove from heat and use the "cold water release" method. Open pan,
stir in cheese and serve.

Date: 07/23/1996

- ☐ Muffins, Pastries, Sco
- ✓ Mushrooms
- ☐ Ostrich
- ☐ Pasta
- ☐ Pies and Tarts
- ☐ Pizza
- ☐ Pressure Cooker
- ✓ Rice
- ☐ Salads
- ☐ San Benito County Co
- ☐ Sandwiches
- ☐ Sauces and Marinades
- ☐ Scheid Vineyards
- ☐ Side Dishes
- ☐ Soups
- ☐ Spain
- ☐ Squash
- ☐ Television Show Recip
- ☐ Thailand
- ☐ Turkey
- ✓ Vegetable Dishes
- ☐ Wine Related Stuff

Record 3/10

Checking and unchecking has no effect on the bizobj, though, since we haven't coded that yet. Let's do that now.

We'll add a chain of responsibility from the dCheckList's onHit() handler, to the form, and then to the bizobj to do the work. Each link in the chain knows its part, and to delegate to the next link appropriately. In CklCategories.onHit(), we get the index of the checklist item that was clicked, and then get the key value of that item. Then we determine, based on that key's presence or absence in the KeyValue list, whether to add that category or remove it. We delegate the adding or removing of the category to the form:

```
def onHit(self, evt):
  idx = evt.EventData["index"]
  idxKey = self.Keys[idx]
  if idxKey in self.KeyValue:
    self.Form.addCategory(idxKey)
  else:
    self.Form.delCategory(idxKey)
```

Add the two new methods addCategory() and delCategory() to FrmRecipes.py:

```
def addCategory(self, catid):
  self.getBizobj().bizReccat.addLink(catid)


def delCategory(self, catid):
  self.getBizobj().bizReccat.delLink(catid)
```

It may seem unnecessary to add such layering. After all, the checklist could have easily gotten the reference to the reccat bizobj as well, and just made those addLink() or delLink() calls without involving the form. However, it didn't take much to add this layer, and now we are ready in the future if we want some other method other than the checklist for the user or program to add or remove category links.

Next, define the new methods, plus a supporting method, in the CategoriesForRecipe bizobj in biz/Reccat.py:

```
def addLink(self, catid):
  if self.hasLink(catid):
    raise ValueError, "catid %s is already linked to this recipe." % catid
  self.new()
  self.Record.catid = catid
  self.save()


def delLink(self, catid):
  if not self.hasLink(catid):
    raise ValueError, "catid %s not found, so can't delete." % catid
  self.seek(catid, "catid")  ## not technically needed
  self.delete()
```

```
58
59
60   def hasLink(self, catid):
61     return self.seek(catid, "catid") >= 0
```

Note that each method first checks if the link already exists or not, and then acts accordingly. If the result isn't what we expect, we raise a ValueError so the programmer (you) knows where the problem is and can find the proper solution. The seek() in line 56 isn't needed, because the seek() in hasLink() will have already placed the record pointer on the relevant record, but we do it anyway in case we change the hasLink() implementation in the future, rendering this assumption false.

I also chose to save() the new record in line 50. Dabo's delete() already commits immediately to the backend, so we may as well make save() immediate in this context, too. It wouldn't make sense to the user that unchecking a category saves the change immediately, while checking a category requires an explicit save().

Cool, we now have the child bizobj showing us the categories and saving changes to category assignments, in an intuitive interface that wasn't all that difficult to code. There were other options for the recipes:categories UI other than the dCheckList, such as a child grid that only listed the linked categories, with an interface to pick categories from some sort of record picker, like another grid showing all the categories. Or we could have used some sort of drag/drop interface. The dCheckList seemed like it would be the easiest for the user to understand and use, so I went with that. When designing things for users, think of the simplest interface that will suffice. Think iPhone, not Visual Studio.

Fixing Up the Select Page

The select page contains generated controls for selecting on different fields. It is a good start for a simple CRUD application, especially for a database administrator maintaining, say, the database for a web app or something. However, the end user will probably find it a bit confusing, as it was written for ease of implementation. Let's collapse the searching on individual fields to just one search box (and search on all fields automatically for the string the user enters). Let's also include the categories list that we already created for the edit page, so the user can click on some categories to filter the results that way.

Here is the new PagSelectRecipes, in its entirety. It introduces some dependencies on things we haven't added yet, but reading through it now will help illuminate why we will be changing existing classes and adding a couple new ones:

```
# -*- coding: utf-8 -*-

import dabo
if __name__ == "__main__":
    dabo.ui.loadUI("wx")
from dabo.dLocalize import _, n_
from PagSelectBase import PagSelectBase
from ChkSelectOption import ChkSelectOption
from CklCategories import CklCategoriesSelect



class PagSelectRecipes(PagSelectBase):

    def setFrom(self, biz):
        biz.setJoinClause("")
        if self.chkCategories.Value:
            # User has enabled category filtering. So dynamically add
the
```

# join clause here.   biz.addJoin("reccat", "reccat.recid = recipes.id", "inner")

```

    def setWhere(self, biz):
        self.super(biz)
        if self.chkCategories.Value:
            # User has enabled category filtering, so build the where
clause
            # on the selected categories.
            whereString = ""
            for catid in self.cklCategories.KeyValue:
                if whereString:
                    whereString += "\n   OR "
                whereString += "reccat.catid = %s" % catid
            if whereString:
                biz.addWhere("(%s)" % whereString)
            else:
                # User chose to filter on categories, but then didn't
choose any:
                biz.addWhere("1=0")

        if self.chkSearch.Value:
            whereString = ""
            for fld in ("title", "subtitle", "ingred", "proced"):
                if whereString:
                    whereString += "\n   OR "
                whereString += "recipes.%s like '%%%s%%'" %
(fld, self.txtSearch.Value)
            biz.addWhere("(%s)" % whereString)


    def getSelectOptionsPanel(self):
        """Return the panel to contain all the select options."""

        panel = dabo.ui.dPanel(self, Sizer=dabo.ui.dSizer("v"))
        gsz = dabo.ui.dGridSizer(VGap=5, HGap=10)
        gsz.MaxCols = 2
        label = dabo.ui.dLabel(panel)
        label.Caption = _("Please enter your record selection
criteria:")
        label.FontSize = label.FontSize + 2
        label.FontBold = True
        gsz.append(label, colSpan=3, alignment="center")

        ## Search all text fields:
        self.chkSearch = ChkSelectOption(panel, Caption="Search:")
        self.txtSearch = dabo.ui.dSearchBox(panel, Name="txtSearch", SaveRestoreValue=True
        self.chkSearch.Control = self.txtSearch

        gsz.append(self.chkSearch, halign="right", valign="middle")
        gsz.append(self.txtSearch, "expand")

        ## Categories:
        self.chkCategories = ChkSelectOption(panel, Caption="Categories:")
        self.cklCategories = CklCategoriesSelect(panel)
        self.chkCategories.Control = self.cklCategories
        dabo.ui.callAfterInterval(200, self.cklCategories.updateChoices)
        dabo.ui.callAfterInterval(200, self.cklCategories.restoreValue)

        gsz.append(self.chkCategories, halign="right")
        gsz.append(self.cklCategories, "expand")

        # Make the last column growable
        gsz.setColExpand(True, 1)
        panel.Sizer.append(gsz, 1, "expand")

        hsz = dabo.ui.dSizer("h")

        # Custom SQL checkbox:
        chkCustomSQL = panel.addObject(dabo.ui.dCheckBox, Caption="Use
Custom SQL",
                OnHit=self.onCustomSQL)
        hsz.append(chkCustomSQL, 1)

        # Requery button:
        requeryButton = dabo.ui.dButton(panel, Caption=_("&Requery"),
                DefaultButton=True, OnHit=self.onRequery)
        hsz.append(requeryButton)

        panel.Sizer.append(hsz, "expand", border=10)
        return panel


if __name__ == "__main__":
    from FrmRecipes import FrmRecipes
    app = dabo.dApp(MainFormClass=None)
    app.setup()
    class TestForm(FrmRecipes):
        def afterInit(self): pass
    frm = TestForm(Caption="Test Of PagSelectRecipes", Testing=True)
    test = frm.addObject(PagSelectRecipes)
    test.createItems()
    frm.Sizer.append1x(test)
    frm.show()
    app.start()
```

Basically, we remove almost all the generated code, and add our own code that explicitly adds a search box (based on dabo.ui.dSearchBox instead of a plain dTextBox) and a slightly different version of CklCategories, called CklCategoriesSelect that doesn't do any data binding. We override some methods of the datanav.PagSelect that PagSelectRecipes defines, in order to add the needed join and where clause additions.

Note that we "OR" the category selection together. In other words, if 'appetizers' and 'mexican' is checked, all appetizers and all mexican recipes will be selected. I can see use cases for either mode, so we'll probably want a way for the user to specify this, such as a dRadioBox or dDropdownList. I'll leave this enhancement as an exercise for the student. For now, if you prefer getting only mexican appetizers, just change that "OR" to "AND" in the addWhere() method.

We refer to a new control, ChkSelectOption, which provides the automatic behavior of enabling/ disabling the associated control when the checkbox is checked or unchecked by the user. Checked criteria will be added to the where clause; unchecked criteria will be ignored. The state of the criteria checkboxes and the associated controls will be saved to the DaboPreferences file for user convenience by setting SaveRestoreValue to True. Define the new class in ui/ChkSelectOption.py:

```
import dabo

class ChkSelectOption(dabo.ui.dCheckBox):
    def initProperties(self):
        self.Alignment = "Right"
        self.SaveRestoreValue = True

    def onHit(self, evt):
        self.setEnabled()

    def setEnabled(self):
        ctrl = self.Control
        if ctrl:
            ctrl.Enabled = self.Value

    def _getControl(self):
        return getattr(self, "_control", None)

    def _setControl(self, val):
        self._control = val
        self.setEnabled()

    Control = property(_getControl, _setControl)
```

Next, we've changed the class definition for the categories check list box. Where there was one class before (CklCategories) there are now three (CklCategoriesBase, CklCategoriesEdit, and CklCategoriesSelect). CklCategoriesBase is never directly instantiated - it exists to define common functionality used by its two subclasses. Here's the complete new listing of CklCategories.py:

```
# -*- coding: utf-8 -*-

import dabo
if __name__ == "__main__":
    dabo.ui.loadUI("wx")

```

```
class CklCategoriesBase(dabo.ui.dCheckList):
    def initProperties(self):
        self._needChoiceUpdate = True


    def updateChoices(self):
        self._needChoiceUpdate = False
        if self.Form.Testing:
            self.Choices = ("Test", "Appetizers", "Greens")
            self.Keys = (1, 2, 3)
            return
        self.Choices, self.Keys = self.Form.getCategoryChoicesAndKeys()


    def update(self):
        self.super()
        dabo.ui.callAfterInterval(200, self.doUpdate)


    def doUpdate(self):
        if self._needChoiceUpdate:
            self.updateChoices()



class CklCategoriesSelect(CklCategoriesBase):
    def initProperties(self):
        self.super()
        self.Width = 200
        self.Height = 250
        self.FontSize = 7
        self.SaveRestoreValue = True



class CklCategoriesEdit(CklCategoriesBase):
    def initProperties(self):
        self.super()
        self.Width = 200


    def doUpdate(self):
        self.super()

        if self.Form.Testing:
            return

        bizRecipes = self.Form.getBizobj()
        bizReccat = bizRecipes.bizReccat
        keyvalues = []
        for rownum in bizReccat.bizIterator():
            keyvalues.append(bizReccat.Record.catid)
        self.KeyValue = keyvalues
    def onHit(self, evt):
        idx = evt.EventData["index"]
        idxKey = self.Keys[idx]
        if idxKey in self.KeyValue:
            self.Form.addCategory(idxKey)
        else:
            self.Form.delCategory(idxKey)



if __name__ == "__main__":
    from FrmRecipes import FrmRecipes
    app = dabo.dApp(MainFormClass=None)
    app.setup()
    class TestForm(FrmRecipes):
        def afterInit(self): pass
    frm = TestForm(Caption="Test Of CklCategories", Testing=True)
    test = frm.addObject(CklCategories)
    frm.Sizer.append(test, 1)
    frm.show()
    frm.update()
    app.start()
```

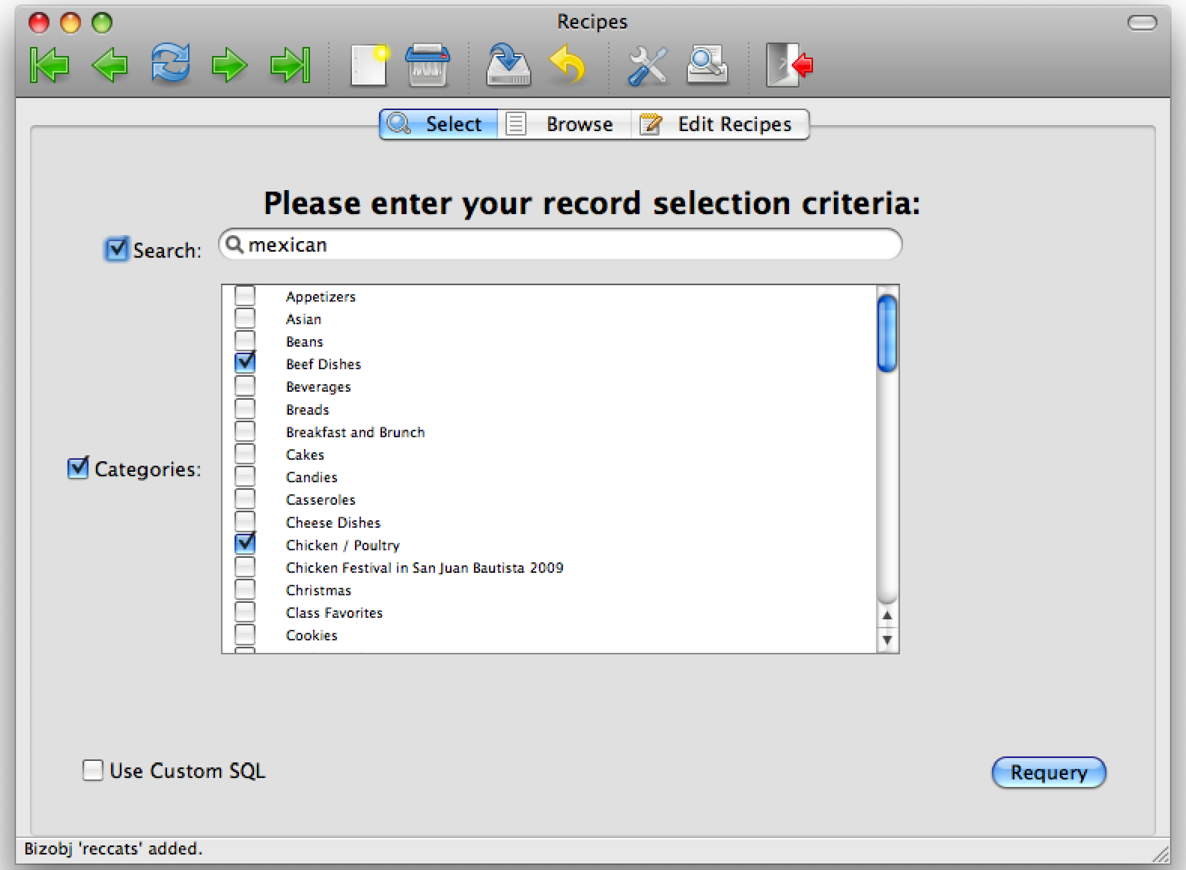
We've encapsulated the data-binding bits into CklCategoriesEdit, while both CklCategoriesEdit and CklCategoriesSelect retain the code to populate the list with all category choices. We've also wrapped the update() cycle into a dabo.ui.callAfterInterval() (line 23-26). This lets the user navigate the recipes faster, since the updating of the checked categories will only happen once, after 200ms has passed. No need for that code to run if the user just wants to navigate around quickly. As soon as they stop on a recipe, the checked categories will be updated. This application is getting more and more user-friendly!

Make the needed changes to the class references in PagEditRecipes.py, by searching and replacing 'CklCategories' with 'CklCategoriesEdit'. I see two instances of this string, on lines 7 and 84.

I've made some other performance-related improvements to FrmRecipes.py. I've removed the requeryCategories() call from the form's afterInit(), and placed it inside some expanded logic of the getCategoryChoicesAndKeys() method. This new version saves a cached version of the choices and keys, so we only ever build that list once, unless we force it to rebuild:

```
def getCategoryChoicesAndKeys(self, forceRequery=True):
    """Return two lists, one for all descrp values and one for all id values."""
    cache = getattr(self, "_cachedCategories", None)
    if not forceRequery and cache is not None:
        return cache
    choices, keys = [], []
    biz = self.getBizobj("reccats")
    if biz.RowCount <= 0 or forceRequery:
        self.requeryCategories()
    for rownum in biz.bizIterator():
        choices.append(biz.Record.descrp)
        keys.append(biz.Record.id)
    self._cachedCategories = (choices, keys)
    return self._cachedCategories
```

With a little work, the default AppWizard-generated application can become much more user-friendly and robust. Here's the recipes select page, after all the above changes:



Please enter your record selection criteria:

[✓] Search: mexican

[✓] Categories:
- [ ] Appetizers
- [ ] Asian
- [ ] Beans
- [✓] Beef Dishes
- [ ] Beverages
- [ ] Breads
- [ ] Breakfast and Brunch
- [ ] Cakes
- [ ] Candies
- [ ] Casseroles
- [ ] Cheese Dishes
- [✓] Chicken / Poultry
- [ ] Chicken Festival in San Juan Bautista 2009
- [ ] Christmas
- [ ] Class Favorites
- [ ] Cookies

[ ] Use Custom SQL

Bizobj 'reccats' added.

Printing out a Recipe

The AppWizard did generate a report form, but it isn't relevant to recipes at all. You can see it by clicking Reports|Sample Report. The menu is defined in ui/MenReports.py. The data is gotten from db/getSampleDataSet.py. The report definition is in reports/sampleReport.rfxml. It is xml, viewable by any text editor or by opening it with dabo/ide/ReportDesigner.py.

We'll create our own report from scratch, and take away access to the sample report, but before we do that let's take a look at one other thing the AppWizard gave you: Quick Reports.

Enter some selection criteria in the Recipes Select page, requery, and then run Reports|Quick Report. Select 'list format', and 'all records in the data set'. Press 'OK'. You'll get a listing of recipe titles and dates in your default PDF viewer. This report was generated from the current data, with a format defined by the current columns in the browse grid. Change the widths of the columns in the browse grid, and the next time you run the quick report the new width will reflect there. Selecting 'expanded format' results in one recipe per page, printing the fields as sized in the edit page.

So these reports are fairly ugly, but they are generated from the browse and edit pages as they exist right now, and they give you something to start with. Let's use the expanded format as a starting point for our recipe printout. Click 'expanded', 'all records', and 'advanced'. Answer 'yes' to the dialog. Now press 'OK'. You now have, in your application's reports/ directory, a new report called 'datanav-recipes-expanded.rfxml'. We can now edit that in ReportDesigner, making it look as pretty as we can imagine, and after we save it from then on the Recipes application will use that report instead of automatically creating the format each time.

(The rest of this section appears in the online version of the tutorial notes.)

Distributing your Application Natively

Dabo doesn't provide any mechanism for building and deploying your application. It doesn't need to: there are already several choices for packaging, or bundling, your application and finally distributing, or deploying it. The packagers all work in to python's setuptools/distutils framework (they work with standard setup.py scripts). Packagers include cx_Freeze, py2app, and py2exe. The AppWizard generates a setup.py file that will use one of the above packagers to build, or compile, your application into native code to run on Linux, Macintosh, and Windows. AppWizard also generates helper scripts in your main directory called 'buildlin', 'buildmac', and 'buildwin.bat'. If you are on Mac and you want to package your application for distribution on that platform, you'd run 'buildmac'. The folder to distribute after running the builder is the 'dist' directory. The application should run on any other computer with an operating system reasonably of the same vintage as the operating system you built from. So, running 'buildwin.bat' on Vista should produce an EXE that runs on Windows7, XP, Vista, and perhaps even Windows2000. Running 'buildmac' on OS X 10.5 should produce a .app that runs on 10.4, 10.5, and 10.6.

For Windows, the AppWizard also includes a skeleton mechanism for making an installer file, so that the user can run a familiar setup.exe to install the application. AppWizard generates a .iss file, and then invokes InnoSetup (if installed) with that .iss file as an argument.

There are a few files that are relevant for customization: App.py, __version__.py, setup.py, and setup.iss.txt. The latter is only relevant for Windows, though.

App.py is your application class definition, subclassed from dabo.dApp. It contains some user setting declarations that are picked up and used by the generated setup.py. Here's how those declarations look, and hopefully it is apparent what type of information should go into each:

```
class App(dabo.dApp):
  def initProperties(self):
```

```
 9     # Manages how preferences are saved
10     self.BasePrefKey = "dabo.app.recipes_aw"
11
12     ## The following information can be used in various places in
your app:
13     self.setAppInfo("appShortName", "Recipes_Aw")
14     self.setAppInfo("appName", "Recipes_Aw")
15     self.setAppInfo("copyright", "(c) 2008")
16     self.setAppInfo("companyName", "Your company name")
17     self.setAppInfo("companyAddress1", "Your company address")
18     self.setAppInfo("companyAddress2", "Your company CSZ")
19     self.setAppInfo("companyPhone", "Your company phone")
20     self.setAppInfo("companyEmail", "Your company email")
21     self.setAppInfo("companyUrl", "Your company url")
22
23     self.setAppInfo("appDescription", "Describe your app.")
24
25     ## Information about the developer of the software:
26     self.setAppInfo("authorName", "Your name")
27     self.setAppInfo("authorEmail", "Your email")
28     self.setAppInfo("authorURL", "Your URL")
29
30     ## Set appVersion and appRevision from __version__.py:
31     self.setAppInfo("appVersion", version["version"])
32     self.setAppInfo("appRevision", version["revision"])
```

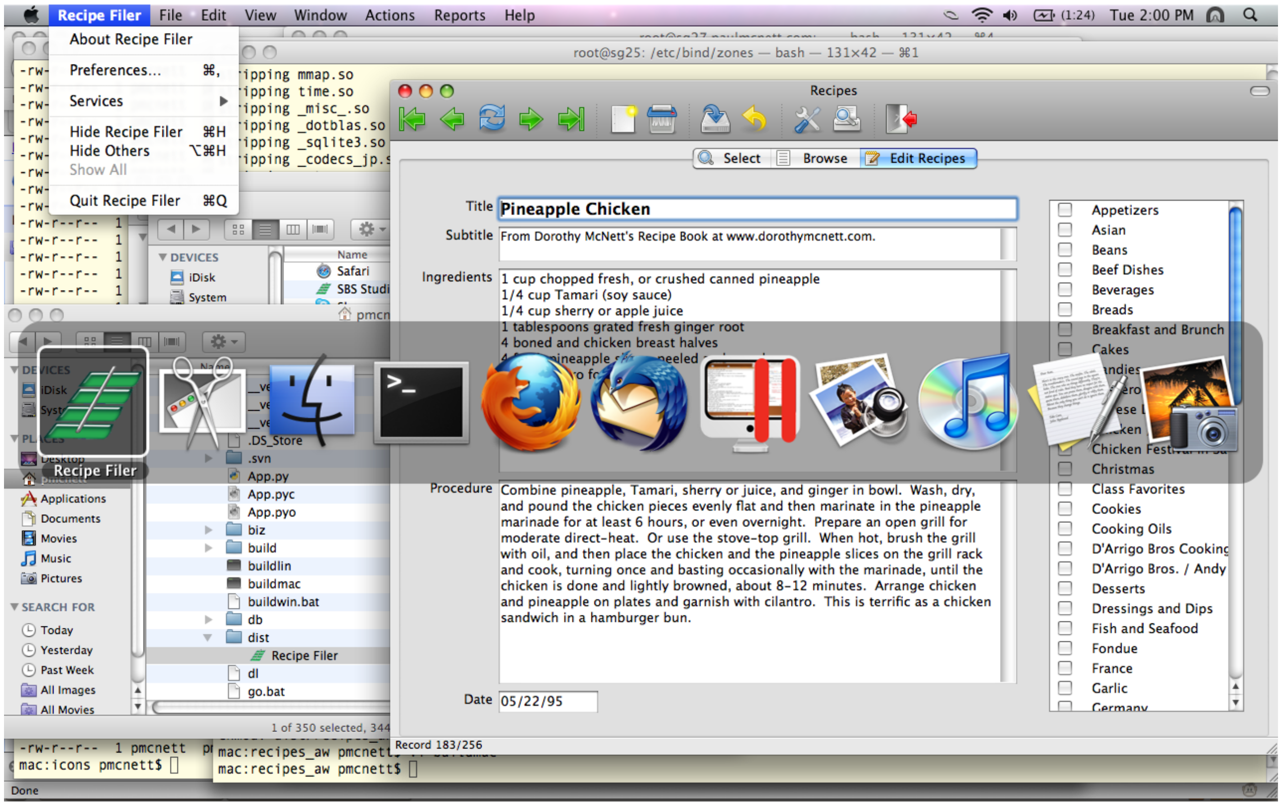
dApp.setAppInfo() saves an arbitrary value to the application instance, using the key given. The only really important piece of information to change here is AppName, since that determines the name of your application at runtime. Change it from "Recipes_Aw" to something like "Recipe Filer".

Line 31 automatically saves the appVersion, for easy reference anywhere in the app: just call self.Application.getAppInfo("appVersion"). The actual version number is encapsulated in the __version__.py file.

There actually isn't much of anything to customize inside of setup.py, because all relevant information is gotten from the application instance. However, if your application starts relying on other libraries, you may find that py2exe, py2app, or cx_Freeze need some help including all the files needed into the distribution. That is what the "includes", "packages", "data_files" and the like are for. The best thing to do is google for the specific answer, or just perform trial-and-error until you get a successful packaging script. Note that there's platform-specific bracketing of the code that calls setup() and builds the options list.

You will want to customize the icon, though. By default, no icon is included, which will result in a platform-specific default icon being used. Icons tend to be platform-specific beasts, and this is beyond the scope of the tutorial, but to give you and example, I've included logo_green.icns from my commercial application in the resources directory of the online source code. This is the Macintosh version of the icon. Simply uncomment the iconfile: declaration in the Mac portion of setup.py. Here's a screenshot of our Recipe Filer running on Mac. Note the use of the icon in the

Finder view, as well as in the task-switcher I've enabled with Cmd-Tab.



```
ipping mmap.so
ipping time.so
ipping _misc_.so
ipping _dotblas.so
ipping _sqlite3.so
ipping _codecs_jp.s
```

```
-rw-r--r--  1
-rw-r--r--  1
-rw-r--r--  1
-rw-r--r--  1
-rw-r--r--  1
-rw-r--r--  1 pmcnett
mac:icons pmcnett$
```

```
mac:recipes_aw pmcnett$
mac:recipes_aw pmcnett$
```

.DS_Store
.svn
App.py
App.pyc
App.pyo
biz
build
buildlin
buildmac
buildwin.bat
db
dist
Recipe Filer
dl
go.bat

1 of 350 selected, 344

Title: **Pineapple Chicken**

Subtitle: From Dorothy McNett's Recipe Book at www.dorothymcnett.com.

Ingredients:
1 cup chopped fresh, or crushed canned pineapple
1/4 cup Tamari (soy sauce)
1/4 cup sherry or apple juice
1 tablespoons grated fresh ginger root
4 boned and chicken breast halves

Procedure: Combine pineapple, Tamari, sherry or juice, and ginger in bowl. Wash, dry, and pound the chicken pieces evenly flat and then marinate in the pineapple marinade for at least 6 hours, or even overnight. Prepare an open grill for moderate direct-heat. Or use the stove-top grill. When hot, brush the grill with oil, and then place the chicken and the pineapple slices on the grill rack and cook, turning once and basting occasionally with the marinade, until the chicken is done and lightly browned, about 8-12 minutes. Arrange chicken and pineapple on plates and garnish with cilantro. This is terrific as a chicken sandwich in a hamburger bun.

Date: 05/22/95

Record 183/256

Creating a Dabo Application From Scratch

While the AppWizard is powerful, it is designed to create a particular type of application. But of course, not every application need can be solved by such a tool, and Dabo is not limited in the types of applications it can create. We're going to build a working application from scratch, and in the process, introduce you to important concepts within the Dabo framework.

The Application

Both Paul and I have made our living as consultants, and as such we created applications for others and billed them for our time. So for this exercise we'll be creating an app to record the time spent for various clients, and then, time permitting, a way to generate bills to send to the clients.

The Data

There are two tables for this app: one to hold information about the clients, and one to record the time spent. In real life there can be many types of billing rates, such as a day rate for travel, but we'll keep it simple and only work with hourly billing, with quarter-hour increments. These two tables are named 'clients' and 'hours', and are in the same 'pycon' MySQL database as the recipe tables we used earlier.

Getting Started

We'll start by determining a directory where we want to create our app. It really doesn't matter where, so follow whatever file organization you normally use. I'm going to use the directory **`~/apps/pycon_hours`** on my drive.

Open up a terminal window (Windows users, use Start|Run, and type 'cmd' for the program name), and change directory to the parent directory for your app; in this example, it would be **`~/apps`**. If I were to run a directory listing, it would look like:

```
[ed@Homer ~]$ cd apps/
[ed@Homer ~/apps]$ ll
total 0
drwxr-xr-x  19 ed  staff   646B Feb  2 12:20 pycon_appwizard
[ed@Homer ~/apps]$
```

So from here, run the following command:

```
[ed@Homer ~/apps]$ python -c "import dabo; dabo.quickStart('pycon_hours')"
[ed@Homer ~/apps]$ ll
total 0
drwxr-xr-x  19 ed  staff   646B Feb  2 12:20 pycon_appwizard
drwxr-xr-x   9 ed  staff   306B Feb  5 15:25 pycon_hours
```

Note how we now have a directory named 'pycon_hours'. Change to that directory, and then do a listing:

```
[ed@Homer ~/apps]$ cd pycon_hours/
[ed@Homer ~/apps/pycon_hours]$ ll
total 8
drwxr-xr-x  3 ed  staff  102B Feb  5 15:25 biz
drwxr-xr-x  3 ed  staff  102B Feb  5 15:25 db
-rwxr--r--  1 ed  staff  282B Feb  5 15:25 main.py
drwxr-xr-x  3 ed  staff  102B Feb  5 15:25 reports
drwxr-xr-x  3 ed  staff  102B Feb  5 15:25 resources
drwxr-xr-x  3 ed  staff  102B Feb  5 15:25 test
drwxr-xr-x  3 ed  staff  102B Feb  5 15:25 ui
[ed@Homer ~/apps/pycon_hours]$
```

This is the main directory structure of a Dabo application, along with a file named **`main.py`** that is used to launch your application. In fact, if you run **`main.py`** right now, you'll get an actual running application. Sure, it's just one empty window and a menu, but it runs! Here's what main.py looks like:

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import dabo
dabo.ui.loadUI("wx")

app = dabo.dApp()

# IMPORTANT! Change app.MainFormClass value to the name
# of the form class that you want to run when your
# application starts up.
app.MainFormClass = dabo.ui.dFormMain

app.start()
```

The script is pretty simple, but it contains some important things to note. First, we import the dabo module, and call the **`dabo.ui.loadUI("wx")`** command to move all of the wxPython-specific code into the dabo.ui namespace.

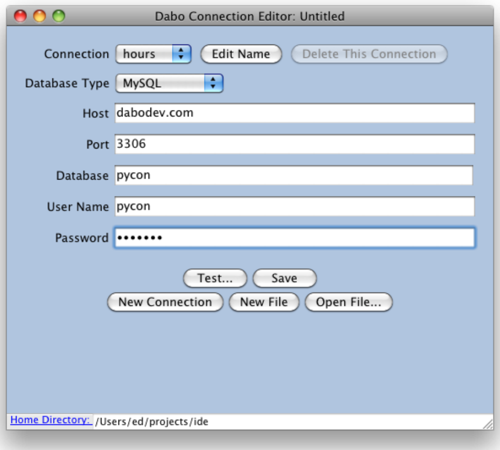
Next we create an application instance. We then set its **MainFormClass** property to the default main form class provided by Dabo. This tells the app which form to show upon startup; when we create our own form for this app, we'll want to change this to our form class.

Finally, we get everything going by calling **`app.start()`**. This will load all the connection definitions, bizobj classes, ui classes, and whatever else you have created, into the app's namespace, display the main form, and then start the main UI event loop. Any code added after this line will not be executed until after the application exits.

The Application object is important in Dabo: if an application is a symphony of all its parts, the Application object is the conductor. To make it easy for you code to refer to this object, we've added a property to all Dabo classes, so that from any object you can refer to the app using **`self.Application`**.

Creating the Database Connection

Dabo has a visual tool for creating and testing database connections. It's called **CxnEditor (short for 'Connection Editor**, and is located, along with all the other visual tools, in the 'ide' directory. Start it up, and you will see a form that looks like the image below. I've filled in the values we will be needing for the tutorial; you can't read the password, but it's 'atlanta'. I gave the connection the name 'hours'; the exact name you give it isn't important, but it's best to give it something descriptive, because your application code will get that connection by referencing that name. You can store multiple connection definitions in a single file, but to be honest, I haven't yet had the need for more than one.



Connection: hours
Database Type: MySQL
Host: dabodev.com
Port: 3306
Database: pycon
User Name: pycon
Password: •••••••

Home Directory: /Users/ed/projects/ide

To make sure that you've typed everything correctly, click the 'Test...' button. You will get a message informing you either of your success, or the reason why the connection failed. When you have it working, click the 'Save' button. It will present you with the standard 'Save As...' dialog for your OS; navigate to the 'db' folder off of the main application folder we created with quickStart, give your file a name such as 'hours', and save. This will create a file named 'hours.cnxml' in the 'db' directory; it's simply an XML file containing the connection information.

You might have noticed the status bar of the above screenshot has what looks like a hyperlink with the text 'Home Directory' on it. This document is being written a few weeks before PyCon, and we're adding some new stuff to our visual tools. In this case, we're adding the concept of Home

Directory to our apps to help with some of the pathing confusion that can arise. Depending on how much time is available for development between now and PyCon, this may or may not be released by then. And yes, that is a Dabo dHyperLink control that is configured to open the standard OS dialog for selecting a directory.

A Note About Encryption

If you save your connection, Dabo creates a file with a '.cnxml' extension. This is a plain XML file, and will look something like this:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<connectiondefs xmlns="http://www.dabodev.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.dabodev.com conn.xsd"
xsi:noNamespaceSchemaLocation = "http://dabodev.com/schema/conn.xsd">

    <connection dbtype="MySQL">
        <name>hours</name>
        <host>dabodev.com</host>
        <database>pycon</database>
        <user>pycon</user>
        <password>Z6DEC5O86ZEEH53Y85O07</password>
        <port>3306</port>
    </connection>


</connectiondefs>
```

This looks pretty straightforward, except for the 'password' element: we had typed 'atlanta' as our password, but Dabo stored an encrypted version in the file. Dabo comes with a very simple encryption/decryption method that really doesn't do much except obscure the actual password from a casual glance, but if anyone wanted to see what your password was, all they would need to do is get a copy of Dabo and run the following:

```
>>> import dabo
>>> app = dabo.dApp()
>>> print "The password is:", app.decrypt("Z6DEC5O86ZEEH53Y85O07")
/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/
site-packages/dabo/lib/SimpleCrypt.py:52: UserWarning: WARNING:
SimpleCrypt is not secure. Please see http://wiki.dabodev.com/
SimpleCrypt for more information
  warnings.warn("WARNING: SimpleCrypt is not secure. Please see
http://wiki.dabodev.com/SimpleCrypt for more information")
The password is: atlanta
```

When you call the default encrypt/decrypt methods, Dabo prints the warning you see above. Yeah, it gets annoying to have that print out all the time, but that's the point: you shouldn't rely on weak encryption. Fortunately, there are two much better alternatives:

PyCrypto

This library is available at http://www.dlitz.net/software/pycrypto/. It has many, many features, but all we're going to use is the DES3 encryption routines. Make sure that the library is installed on your machine, using pip, easy_install, or a manual setup.

Once PyCrypto is installed, you only need to set the Application object's **CryptoKey** property. This should be either a string, or, for better security, a callable that will return the string. Remember, the best encryption is pretty worthless if you you leave your encryption key around in a plain text file where just about anyone can read it.

Your Own Custom Encryption Object

For higher security requirements you should provide your own custom class to handle all encryption/decryption. The class must expose two methods: **encrypt()** and **decrypt()**. Then you need to set the Application object's **Crypto** property to an instance of your custom cryptography class, and Dabo will use that object's routines instead of its own.

Creating the Bizobj Classes

Now that we have our connection defined and working, the next thing to do is create the bizobj classes so that our app can access that data. We're going to create two classes: one for the 'clients' table, and one for the 'hours' table. We'll start with the clients table, and create that one by hand; we'll create the hours class with one of the visual tools.

Using your favorite text editor, create a file in the 'biz' directory of your application's folder named 'ClientBizobj.py'. The code will look like this:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import dabo

class ClientBizobj(dabo.biz.dBizobj):
    def afterInit(self):
        self.DataSource = "clients"
        self.KeyField = "pkid"
        self.addFrom("clients")
        self.addField("pkid")
        self.addField("clientname")
        self.addField("attn")
        self.addField("rate")
        self.addField("street1")
        self.addField("street2")
        self.addField("city")
        self.addField("stateprov")
        self.addField("postalcode")
```

Here we're using the standard SQL Builder approach to defining the bizobj's data structure. We start by importing **dabo**, and inherit our class from **dabo.biz.dBizobj**, which is the base bizobj class. We add our code to the **afterInit()** method, and start by defining the most important properties: **DataSource** and **KeyField**. After that, we add all the fields from the table that we want to use in our app; in this case, it's all of the fields. We can add more logic to the class later, but this is enough to get us started.

We also need to add this class into the 'biz' module's namespace; we do this by editing the **__init__.py** file (which was created by quickStart) in the biz directory. Add the line at the end so that the file reads:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
######
# In order for Dabo to 'see' classes in your biz directory, add an
# import statement here for each class. E.g., if you have a file named
# 'MyClasses.py' in this directory, and it defines two classes named
'FirstClass'
# and 'SecondClass', add these lines:
#
# from MyClass import FirstClass
# from MyClass import SecondClass
#
# Now you can refer to these classes as:
self.Application.biz.FirstClass and
# self.Application.biz.SecondClass
######

from ClientBizobj import ClientBizobj
```

When we create the HoursBizobj later, we'll add that to this file, too.

Creating the User Interface

We're going to create a form that will be used to enter our billable hours. It will also show all of the unbilled hours we've entered, and give us a way to create bills for our clients. We'll start by creating the data entry form first.

From the home directory in your app, you'll want to run the following command:

```
/path/to/ClassDesigner.py ui/hours.cdxml
```

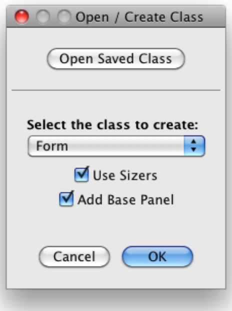
Be sure to substitute the actual path to your 'ide' directory. This can be a pain to type all the time, so on my system I create an alias:

```
alias des='/Users/ed/projects/ide/ClassDesigner.py'
```

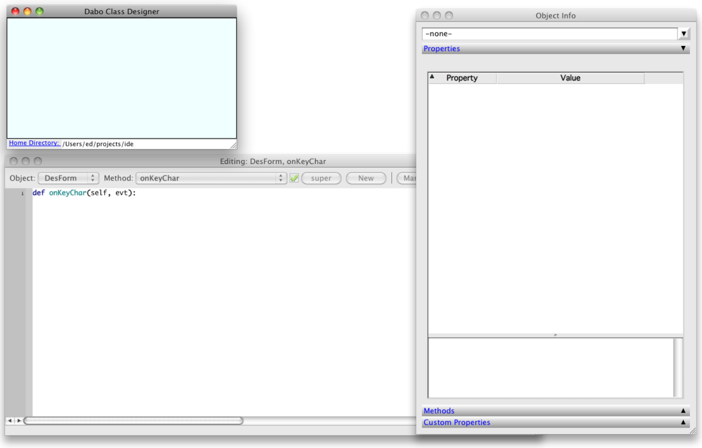
and then I can just type:

```
des ui/hours.cdxml
```

When you run that command, you'll get a dialog informing you that the file doesn't exist, and asking if you wish to create it. Answer yes, and then you'll see a dialog asking what type of class you wish to create (below left). One thing to note is that the Class Designer is not just for form classes; you can create specialized UI component classes that you can re-use in multiple forms and projects. But for now just accept the defaults; you'll then see three windows appear: the Class Designer design surface, the Object Info window, and the Code Editing window (below right).

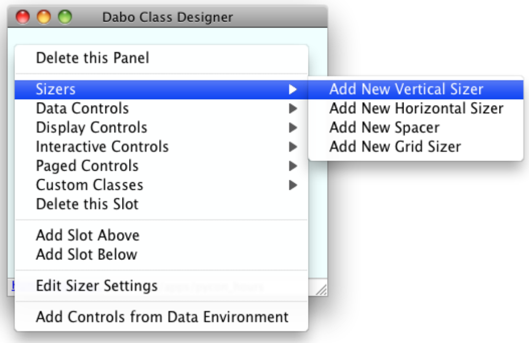


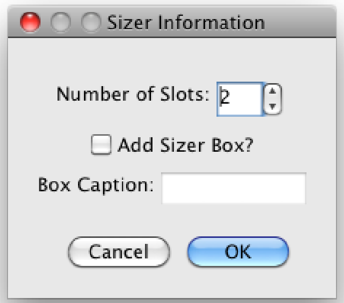
Select the class to create: Form
☑ Use Sizers
☑ Add Base Panel



```
def onKeyChar(self, evt):
```

Start by resizing the design surface window to be bigger, and then right-click on it. This is the primary way of adding or changing design surface items. You'll get a popup that looks like the image below left. Note that the controls are broken down into various categories to make finding the one you want a little easier; go ahead and browse through those options to get an idea of the various controls available. But for now, we want to add a vertical sizer to divide the form into two sections. When you select the 'Add New Vertical Sizer' option, you'll see a dialog like the one below right.



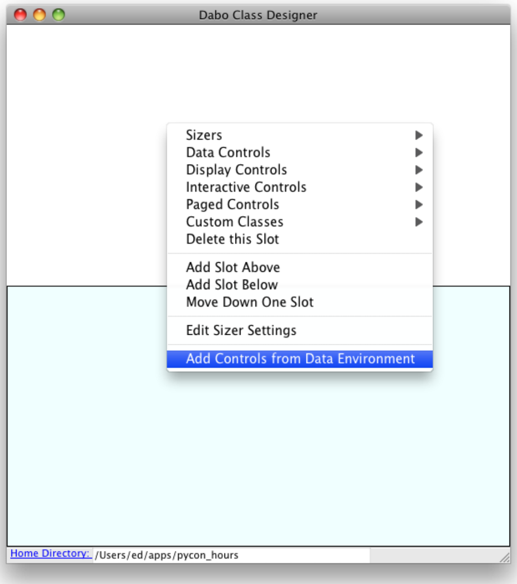


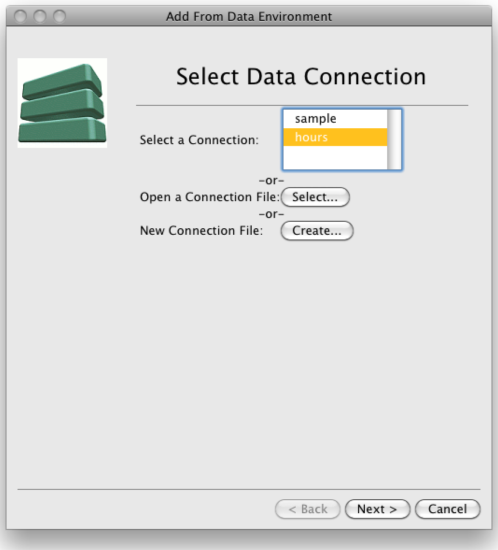
Number of Slots: 2
☐ Add Sizer Box?

Change the 'Number of Slots' to 2, and click 'OK'. You'll now see that the form is divided vertically into two even sections. We'll talk a little more about sizers in a bit, but for now, just be aware that they are basically rules for laying out the items on a form. We're going to add the controls for the data entry to the top section, and the listing of unbilled hours to the bottom half.

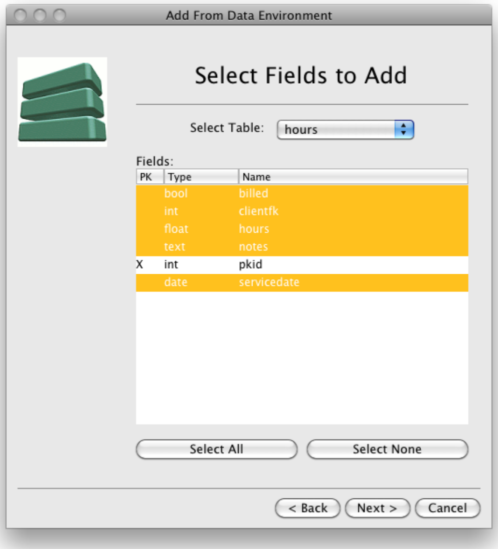
Using the Data Environment Wizard

Right-click on the top half, and select the bottom-most option (below left). We could add the controls for each field in the Hours bizobj one by one, but this will make it faster and less prone to mistakes. It will also create the Hours bizobj for us! After you select this option, you will see the Data Environment wizard (below right).





You should see the 'hours' connection, which Dabo automatically found when you started up the Class Designer from your app's home directory; if you don't see it, you can click on the 'Open a Connection File' option and select that file. There is an option to create a new connection, but it only seems to work consistently on Mac and Windows. You might also see a connection named 'sample', as in the screenshot above; this is a sample connection file that is included in the 'ide' directory, so Dabo will usually find that one, too, when running the visual tools. Select 'hours', and click 'Next'. You should see something like the image on the left below. The table that initially shows will probably be 'clients', since this is sorted alphabetically. Change that to 'hours', and you should see the columns in the hours table displayed. We want all of them except for the PK, since we don't need to display the key field value. Click the 'Select All' button, and then control-click (command-click on Mac) the 'pkid' row to unselect it. Click the 'Next' button; you will now be given an opportunity to arrange the order that these fields are displayed. To move them, click their name to select, and then click the up/down arrows to move them in the list. When you're done it should look like this:

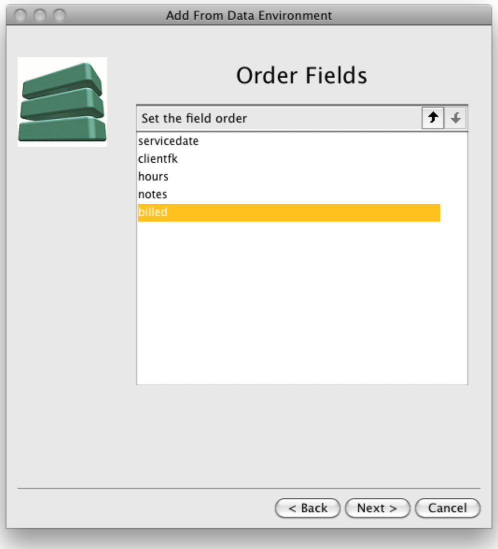

Add From Data Environment

Select Fields to Add

Select Table: hours

Fields:

PK	Type	Name
	bool	billed
	int	clientfk
	float	hours
	text	notes
X	int	pkid
	date	servicedate

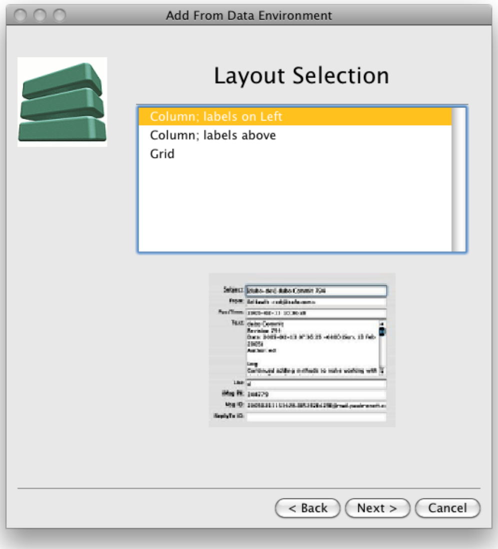

Add From Data Environment

Order Fields

Set the field order

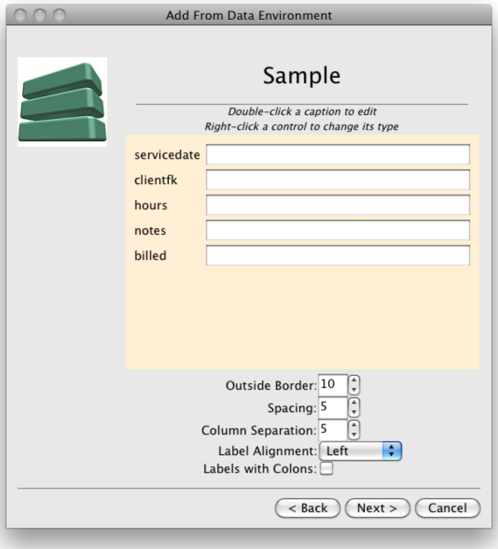
servicedate
clientfk
hours
notes
billed

Click 'Next', and you'll be presented with a choice of layouts (below left). The default first choice is what we want, so click 'Next'. You will now see a preview of layout that the wizard will create (below right). We can customize this before we add the controls to the form; it's easier to do it here, since this is a live preview, and when we make changes, the preview updates to reflect those changes.


Add From Data Environment

Layout Selection

Column; labels on Left
Column; labels above
Grid


Add From Data Environment

Sample

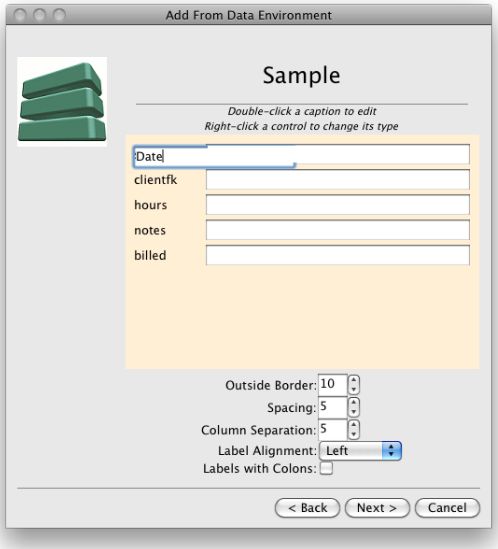
Double-click a caption to edit
Right-click a control to change its type

servicedate
clientfk
hours
notes
billed

Outside Border: 10
Spacing: 5
Column Separation: 5
Label Alignment: Left
Labels with Colons: ☐

Let's start by customizing the labels. They default to the column names in the database, but that's not very user-friendly. To change them, simple double-click the label and you will get an editing box. For example, double-click on the 'servicedate' label, and enter the text 'Date' instead. You'll see something like the image on the left below. Press 'Enter' when you're done, and the label has now been changed! Also note that the layout has adjusted to reflect the smaller label - that's sizers in action! Change the remaining labels to make them look nicer. Also, try changing the various options at the bottom of the preview window: as you change them, the preview updates to reflect the changed settings.

One thing that's pretty obvious is that every field has a text box to hold the value, but they're not all text fields! You can change the control type for each field here, too, by right-clicking on the text box. You'll be presented with a few basic control types: date text box, edit box, spinner and check box. In our case, we want a date text box for the servicedate field; a spinner for the hours field, an edit box for the notes field, and a check box for the billed field. Make those changes, and the preview should now look like the image below right.



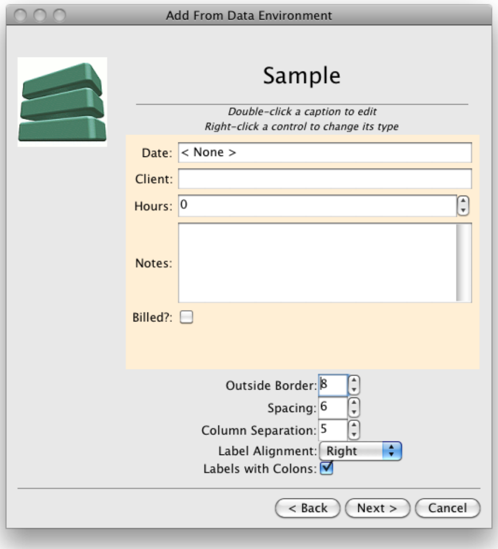
Add From Data Environment

Sample

Double-click a caption to edit
Right-click a control to change its type

Date
clientfk
hours
notes
billed

Outside Border: 10
Spacing: 5
Column Separation: 5
Label Alignment: Left
Labels with Colons: ☐



Add From Data Environment

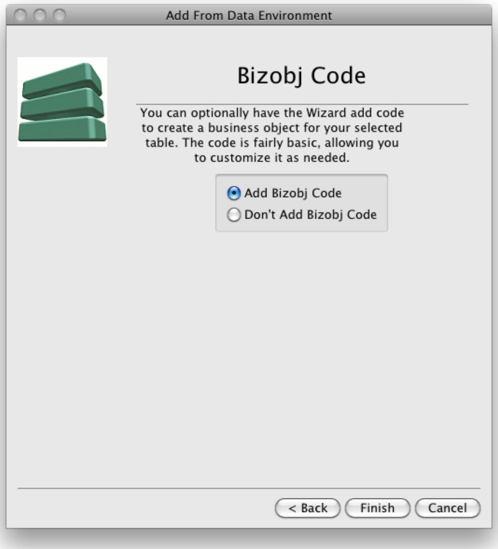
Sample

Double-click a caption to edit
Right-click a control to change its type

Date: < None >
Client:
Hours: 0
Notes:
Billed?: ☐

Outside Border: 8
Spacing: 6
Column Separation: 5
Label Alignment: Right
Labels with Colons: ✓

The controls still aren't *exactly* as we'd like, but we can tweak them later. When you have the preview looking like this, click 'Next'. You'll see the final wizard page (below left). If we had already created our bizobj, we'd tell the wizard not to add it again, but in this case we want it to create the bizobj. So leave the default choice, and click 'Finish'. You will see the controls added to the top half of your form, and they should look something like the image on the right.

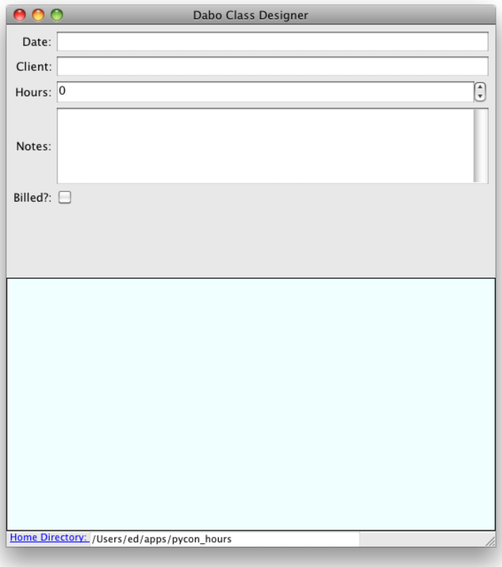


Add From Data Environment

Bizobj Code

You can optionally have the Wizard add code to create a business object for your selected table. The code is fairly basic, allowing you to customize it as needed.

[✓ Add Bizobj Code] Don't Add Bizobj Code



Dabo Class Designer

Date:
Client:
Hours: 0
Notes:
Billed?: ☐

Home Directory: /Users/ed/apps/pycon_hours

This would be an excellent time to save your design before proceeding. In the future I won't remind you to save; just remember to do it every few minutes so that you don't lose too much of your work.

Let's check to make sure that the bizobj was created correctly. If we go back to do a directory listing of the app's biz directory, we now see:

```
[ed@Homer ~/apps/pycon_hours]$ ll biz/
total 48
-rw-r--r--@ 1 ed  staff   438B Feb  6 08:03 ClientBizobj.py
-rw-r--r--  1 ed  staff   871B Feb  6 08:32 ClientBizobj.pyc
-rw-r--r--  1 ed  staff   559B Feb  6 08:32 HoursBizobj.py
-rw-r--r--  1 ed  staff   1.0K Feb  6 08:32 HoursBizobj.pyc
-rw-r--r--@ 1 ed  staff   577B Feb  6 08:32 __init__.py
-rw-r--r--  1 ed  staff   221B Feb  6 08:32 __init__.pyc
```

Let's open up HoursBizobj.py in a text editor. It should look like:

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import dabo

class HoursBizobj(dabo.biz.dBizobj):
    def afterInit(self):
        self.DataSource = "hours"
        self.KeyField = "pkid"
        self.addFrom("hours")
        self.addField("clientfk")
        self.addField("hours")
        self.addField("notes")
        self.addField("billed")
        self.addField("servicedate")
        self.addField("pkid")

    def validateRecord(self):
        """Returning anything other than an empty string from
        this method will prevent the data from being saved.
        """
        ret = ""
        # Add your business rules here.
        return ret
```

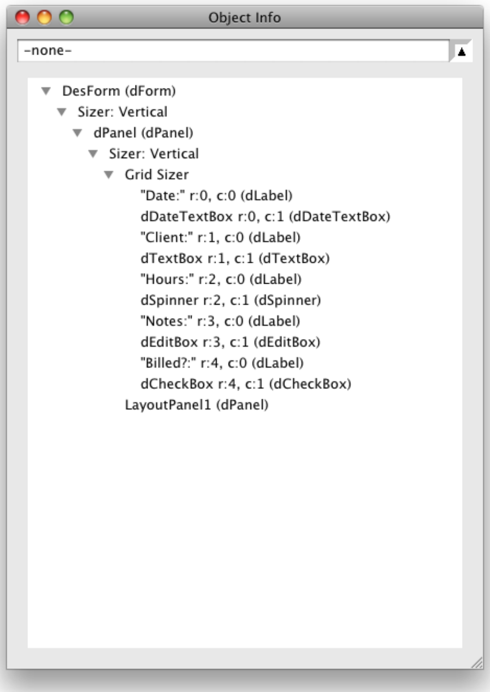
Sure looks an awful lot like the bizobj we hand-coded, doesn't it? In addition, it added a stub **`validateRecord()`** method for us to later add our business logic. Note also that it added the '**pkid**' column, even though we did not select it in the wizard. The reason is that the PK value has to be present in the data, even if it isn't displayed in the UI. The wizard is smart enough to handle that.

Before we leave the editor, open up **`biz/__init__.py`**. Hey, look! The wizard was also smart enough to add the import statement for the HoursBizobj class for us!

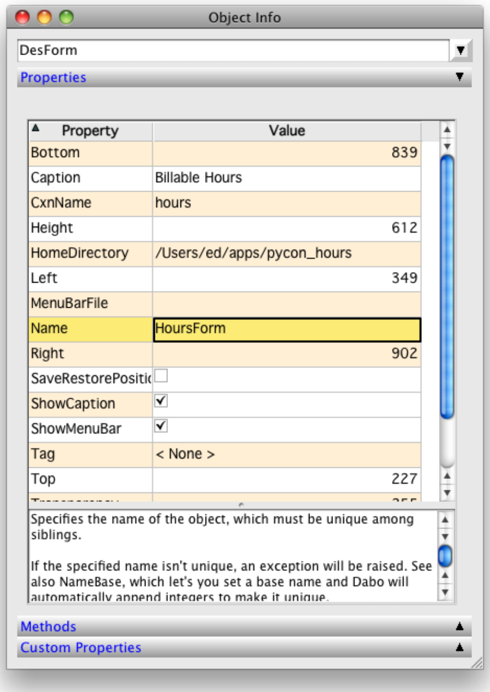
Improving the Appearance

The wizard got us awfully close to what we need, but we can do better. First, let's start by changing the form's Caption to something better than the default 'Dabo Class Designer'. To do that, we need to select the form itself, but that's kind of hard, as we have lots of other stuff on top of it. The way to do this is to go to the Object Info window, and click the black triangle in the top-right corner. That will hide the property sheet and show the Object Tree. The Object Tree is a handy way of visualizing the nested relationship of both the object containership hierarchy as well as the sizer hierarchy. In fact, since sizers are not actual objects, the only way to access them in the Class Designer is via the Object Tree (below left). You should see an entry for 'DesForm' at the top; double-click it to select the form and return to the prop sheet. Click on the right column of the row labeled 'Caption', and change the text to 'Billable Hours'. Press 'Enter' to accept the change, and note the the title bar of the form has now changed to reflect the new Caption. Changes made in the property sheet are instantly reflected in the design surface, making it easy to verify that you did things correctly.

While we're here, let's also change the Name property of the form to something other than the default 'DesForm'. It's not critical that we do so, but it makes things look better. Also, we can be editing multiple classes at once, but there is only one prop sheet, and this helps us see which form the prop sheet is currently reflecting. So change the Name property to 'HoursForm'; the prop sheet should now look like the image below right.



```
▼ DesForm (dForm)
  ▼ Sizer: Vertical
    ▼ dPanel (dPanel)
      ▼ Sizer: Vertical
        ▼ Grid Sizer
            "Date:" r:0, c:0 (dLabel)
            dDateTextBox r:0, c:1 (dDateTextBox)
            "Client:" r:1, c:0 (dLabel)
            dTextBox r:1, c:1 (dTextBox)
            "Hours:" r:2, c:0 (dLabel)
            dSpinner r:2, c:1 (dSpinner)
            "Notes:" r:3, c:0 (dLabel)
            dEditBox r:3, c:1 (dEditBox)
            "Billed?:" r:4, c:0 (dLabel)
            dCheckBox r:4, c:1 (dCheckBox)
          LayoutPanel1 (dPanel)
```



Property	Value
Bottom	839
Caption	Billable Hours
CxnName	hours
Height	612
HomeDirectory	/Users/ed/apps/pycon_hours
Left	349
MenuBarFile	
Name	HoursForm
Right	902
SaveRestorePositic	☐
ShowCaption	✓
ShowMenuBar	✓
Tag	< None >
Top	227

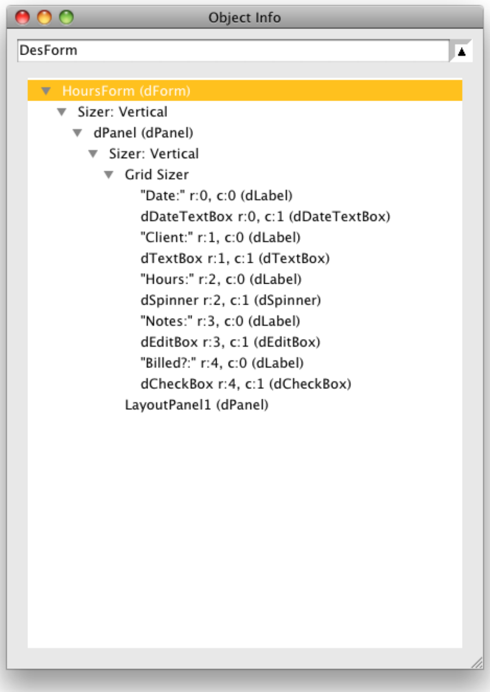
Specifies the name of the object, which must be unique among siblings.

If the specified name isn't unique, an exception will be raised. See also NameBase, which let's you set a base name and Dabo will automatically append integers to make it unique.

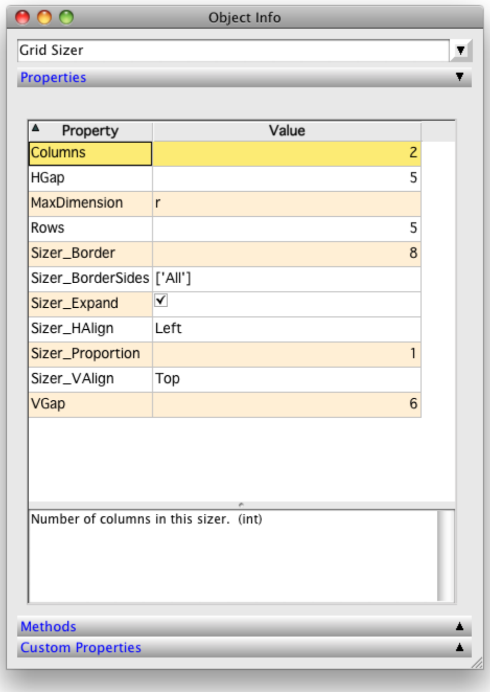
The label at the very top will not refresh immediately, but if you look at the object tree, it has been changed there (below left). Selecting any other object and then re-selecting the form will refresh the label at the top.

I'd like to take a moment here to explain a couple of things about the Property Sheet, or 'prop sheet'. It lists all of the properties of the selected control(s), and allows you to change those that are not read-only. Also, as you navigate through the properties in the grid, the bottom section displays the docstring for that property. You might also notice that there are two other 'sheets' available, the Methods Sheet and the Custom Properties Sheet. Those are beyond the scope of this tutorial, but they provide additional ways to customize your UI classes.

OK, so we've got the form's title bar fixed. What else could be improved? For one, the controls look kind of squished against the edges of the form. We specified a border of 8 pixels in the DE wizard, but apparently that wasn't enough. If we look at the object tree above, note that all of the controls that we added are inside of a 'Grid Sizer': this is a sizer that arranges its objects in a 2-D, grid-like layout. So open the object tree and double-click the 'Grid Sizer' entry. You should see a property sheet that looks like the image below on the right.



```
▼ HoursForm (dForm)
  ▼ Sizer: Vertical
    ▼ dPanel (dPanel)
      ▼ Sizer: Vertical
        ▼ Grid Sizer
            "Date:" r:0, c:0 (dLabel)
            dDateTextBox r:0, c:1 (dDateTextBox)
            "Client:" r:1, c:0 (dLabel)
            dTextBox r:1, c:1 (dTextBox)
            "Hours:" r:2, c:0 (dLabel)
            dSpinner r:2, c:1 (dSpinner)
            "Notes:" r:3, c:0 (dLabel)
            dEditBox r:3, c:1 (dEditBox)
            "Billed?:" r:4, c:0 (dLabel)
            dCheckBox r:4, c:1 (dCheckBox)
          LayoutPanel1 (dPanel)
```



Property	Value
Columns	2
HGap	5
MaxDimension	r
Rows	5
Sizer_Border	8
Sizer_BorderSides	['All']
Sizer_Expand	✓
Sizer_HAlign	Left
Sizer_Proportion	1
Sizer_VAlign	Top
VGap	6

Number of columns in this sizer. (int)

Note that there are a group of properties that all begin with 'Sizer_'; these are not actual properties, but are a handy shortcut to change the properties of the sizer that manages this control's layout. In this case, we want a bigger border around this group of controls, so change the Sizer_Border property to something bigger, such as 30. Note that as soon as you press Enter, the border around all of the controls expands accordingly.

OK, I think we've gone far enough without explaining this whole sizer thing. It's really quite simple once you understand the basic idea, but if you've never used them before, it can take some time to learn to let go of absolute positioning and work with a sizer-based approach. We'll get back to customizing the UI in a little bit; for now, let's talk about sizers.

Sizers

If you've created UIs before, you're probably used to positioning your controls absolutely: you specify a left/top position and a size, and the control always appears in that exact position and that exact size. That works well enough for some things, but has a number of limitations. For example, if the user resizes the form, we'd expect the controls to reflect the new size. Or if a user has a hard time reading and bumps up the font size to compensate. Or if you run the form under multiple OSs, or even multiple themes on the same OS. Under all of these conditions, your carefully laid-out form now looks like garbage.

Sizers present an alternate way of positioning your controls: they are rules that take into account the native size of the control, other controls within the sizer, and the environment in which the sizer exists. There are two main types of sizers: 1-D sizers that lay out controls along a vertical or horizontal dimension, and 2-D sizers that use a grid-based layout. In Dabo, these are **dSizer** and **dGridSizer**, respectively. Sizers can be nested inside of other sizers, as we've already seen: we had a grid sizer that laid out our controls, and it was inside of a vertical sizer.

Sizers are not actual objects, but sometimes it's useful to think of them that way in order to get a clearer picture of how your form is going to work. In the Class Designer you can see this representation in the Object Tree.

Let's start with the simpler 1-D sizers. Objects added to a 1-D sizer can have several settings that affect how the sizer treats them:

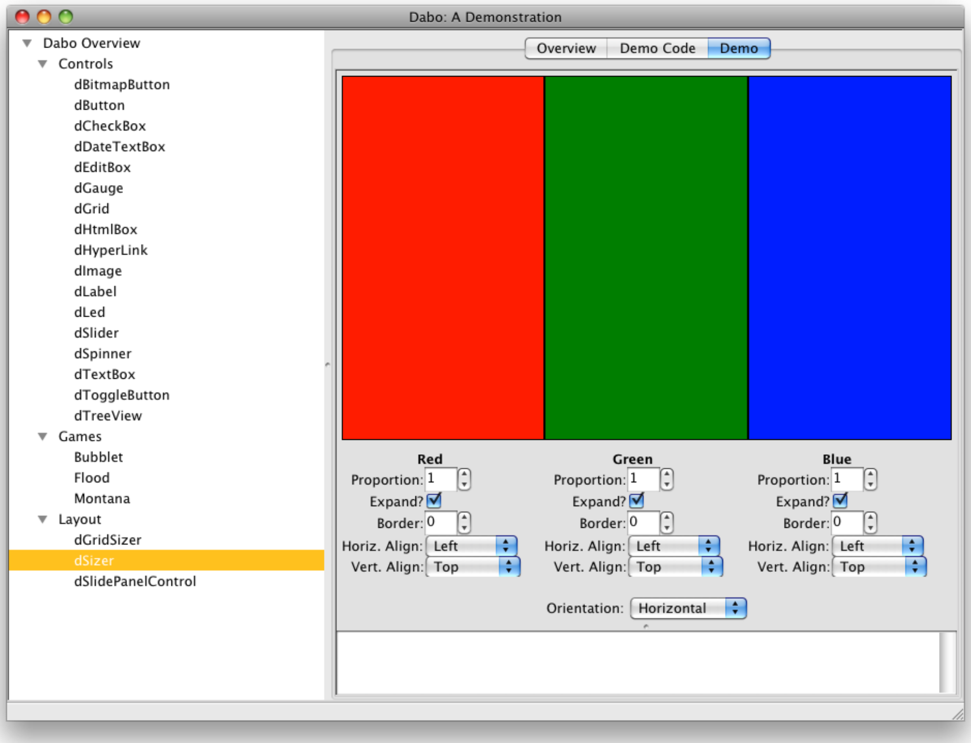
- **Border**: the number of pixels added as a buffer around the control when computing the control's size and position. Default=0.
- **BorderSides**: to which sides of the control is the border applied? Default="all", but can also be a list of any of "top", "bottom", "left" or "right".
- **Proportion**: after the basic widths of controls has been allocated, what proportion of the left over space does this control get?
- **Expand**: does the control expand to fill the "other" dimension?
- **HAlign**, **VAlign**: this controls where the control is positioned along the "other" dimension. By this I mean if this is a vertical sizer, the halign setting would be relevant, and could be one of "left", "center", or "right". For horizontal sizers, the valign settings would be one of "top", "middle", or "bottom".

In order to understand how these affect the control, let's review how sizers work. First, when a resize event is received by the form, it calls the main sizer's layout() method. You can also call layout() manually if your code modifies the controls in the form to update the form's appearance. The sizer is told how much space it has to fill, and it starts by computing the base size of each element it manages. This is the normal displayed size for a control if it existed by itself outside of the sizer. Think of a label containing some text in a certain font: that label needs a certain base size to properly display that text in that font. This is the control's base size. The sizer then adds any border values to each control to get a total basic size for all of its elements. If the size available is smaller than that, the controls are clipped and only some may be visible. But if there is enough size, the controls are each given their basic size plus border. Now, if there is any more space available in the sizer's orientation (i.e., vertical or horizontal), that remaining space is divvied up according to the relative weighting of the controls' proportion settings. If a control's proportion is zero, it gets no additional space. Otherwise, the sizer totals up all the proportions for all of its elements, and gives each control a percent of the extra space according to its proportion.

Using words like the above can be very confusing, so let's take an example: imagine that we have a form that is 200 pixels wide by 300 pixels tall, containing three controls, each 50 pixels wide by 20 pixels tall. The form has a vertical sizer, and these controls are all added to this sizer. The first has a proportion of 0; the second has a proportion of 1, and the third has a proportion of 2. All have a border of zero, so the total base size in the vertical dimension is 60px (3 controls x 20px high each). The total vertical space available is 200px, which leaves 240px left over. The total of the proportions of the elements is 3 (0+1+2), so the sizer would make the first control 20px tall (20px base + (0/3 *240)). The second control would now be (20px base + (1/3 *140)), or 100px tall, and the third would be (20px base + (2/3 *240)) or 180px tall. Adding these up, we get 20 + 100 + 180, or the full 300px available in the form's height.

But what about width? Well, it depends on the setting of Expand: if it's True, the control will expand to the full width of the form, or 200px. If it's False, the width will remain its normal 20px, and its position will be determined by its halign setting.

This is much easier to see visually. There is an excellent sizer demo as part of the DaboDemo app; you can play around with that to get a better feeling for how changing the various settings affects the size and position of controls within a sizer.



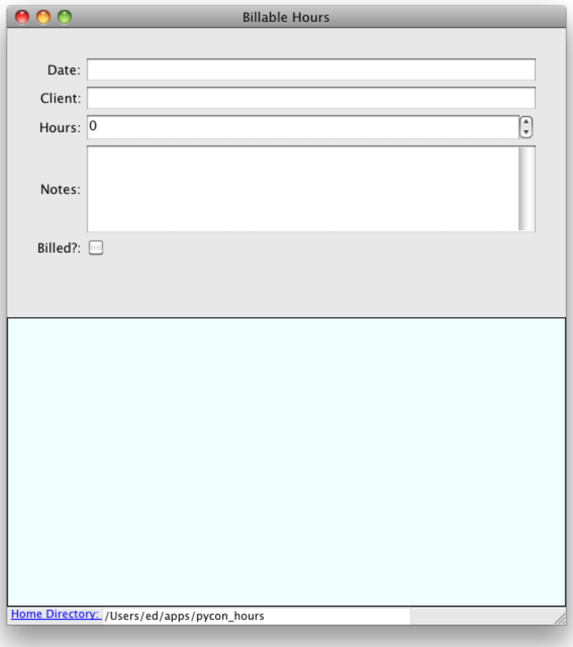
Red	Green	Blue
Proportion: 1	Proportion: 1	Proportion: 1
Expand? ✓	Expand? ✓	Expand? ✓
Border: 0	Border: 0	Border: 0
Horiz. Align: Left	Horiz. Align: Left	Horiz. Align: Left
Vert. Align: Top	Vert. Align: Top	Vert. Align: Top

Orientation: Horizontal

Grid sizers work much the same, with a few important differences. For one, all the controls in a given row have the same height, and all in the same column have the same width. As a result, the settings for **RowExpand** and **ColExpand** will be the same for all controls in a given row or column; changing one affects them all. Different rows and different columns can have different heights and widths, respectively, though. The 'proportion' setting doesn't have any effect, but there are two settings that are available that can make for interesting layouts: **RowSpan** and **ColSpan**. What these do is tell the grid sizer that the control should be spread across multiple rows or columns when calculating its size and position. This is a little beyond the scope of this tutorial, but I thought I'd mention it here.

Back to the Form

OK, while I'm sure that at this point you're not entirely comfortable with all the concepts of working with sizers, that should give you enough background to understand the ideas as we proceed further. To refresh your memory, here's what the form looks like right now:

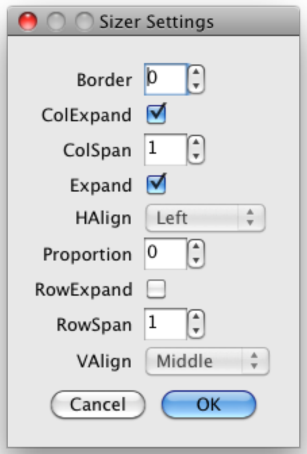


There are a few things that are visually "off": The spinner for the Hours value is too wide; it makes no sense for it to span the full width of the form. The 'Notes:' label is centered vertically; I'd prefer to have it even with the top of its control. And finally, having the label for the checkbox arranged like that is not normal; typically, the label is part of the checkbox itself. We'll address all of these now.

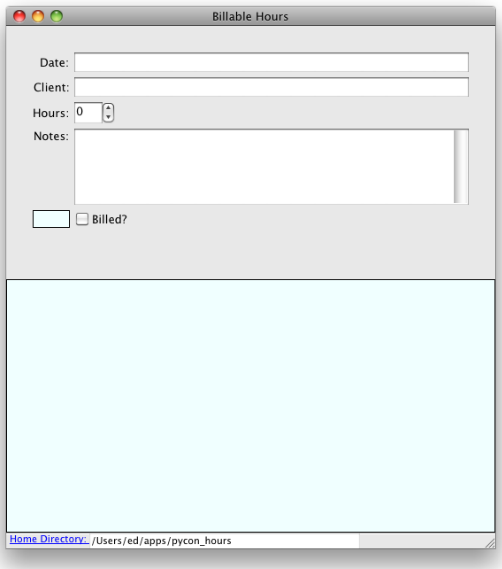
To fix the spinner, right-click on it, and select the item 'Edit Sizer Settings' from the context menu. You'll see a dialog box the looks like the image below on the left. Uncheck the box labeled 'Expand', and you should see the spinner instantly shrink to a more normal size. This sizer editing dialog is a handy way to play around with sizer concepts: you can change any of the properties you like and see the effect of that change right away, and if you don't like what you've done, just click 'Cancel' and the control will revert to its original appearance.

To fix the 'Notes:' label, right click on it and bring up the sizer editing dialog. Change the 'VAlign' setting from 'Middle' to 'Top'. You may want to do this with the other labels as well, as they are also VAligned to middle - it just isn't as noticeable since the dTextBoxes are about the same height as the dLabels.

To change the 'Billed?' checkbox, you'll need to do two things. First, click on the checkbox and set its Caption to 'Billed?'. Then right-click on the label and select 'Delete'. You'll see a rectangular "hole" where the label was; this is a visual "slot" that indicates an empty position in the sizer. At runtime it will be replaced by a blank panel, so you can leave it as is. When you've done this, the form should look like the image below right.


Sizer Settings

Border 0
ColExpand ✓
ColSpan 1
Expand ✓
HAlign Left
Proportion 0
RowExpand ☐
RowSpan 1
VAlign Middle

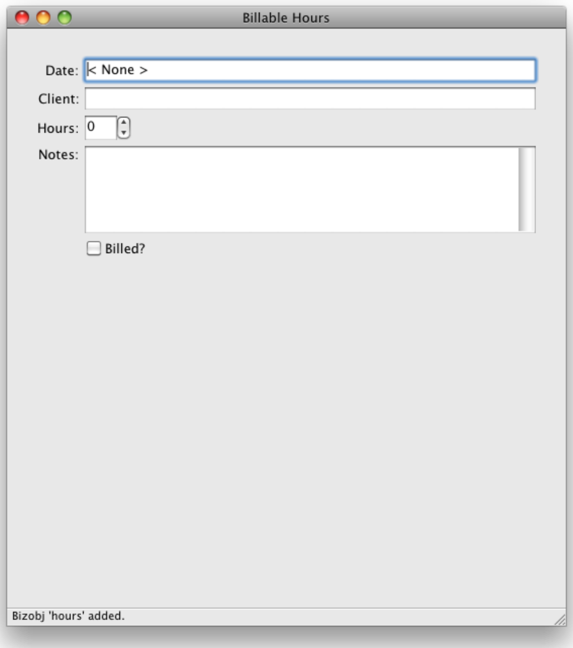

Billable Hours

Date:
Client:
Hours: 0
Notes:
☐ Billed?

Home Directory: /Users/ed/apps/pycon_hours

Taking a Test Drive

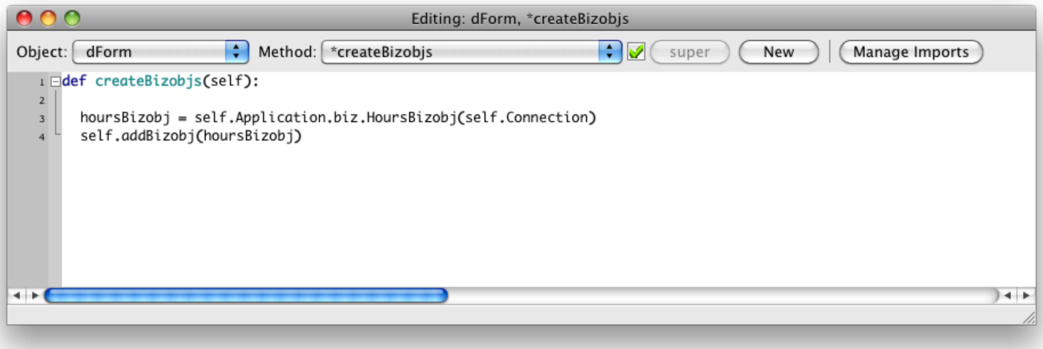
You can do some quick visual testing of your form right from the Class Designer. Select 'Run...' from the File menu, and you should see this:



Doesn't look all that different, does it? But this is an actual live form created from your class design. Drag it around, and you'll still see your design surface underneath. Resize the form, and note that the control resize accordingly. There is no data yet, since we haven't told it to load any. Let's fix that now.

Connecting the Bizobjs

Close the live form, and go to the code editing window. This is where we add code to our class. You might notice that there is already some code there:



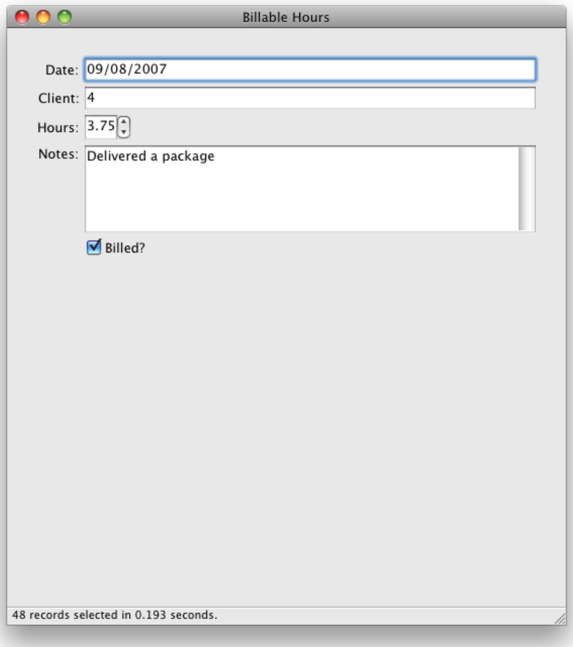
```
def createBizobjs(self):

    hoursBizobj = self.Application.biz.HoursBizobj(self.Connection)
    self.addBizobj(hoursBizobj)
```

The code is in the form's **createBizobjs()** method, which is automatically called by the framework during the form's creation. The code was added by the DE Wizard when we ran it to add the controls. Let's look at the code a little closer.

First we instantiate the bizobj for the hours table, using the class reference **self.Application.biz.HoursBizobj**. How did it get that class definition and that namespace? If you recall, all Dabo objects can refer to the main app object via the **self.Application** reference; during the app startup, the app object will populate its 'db', 'biz', etc. (i.e., all the standard subdirectories) attributes with the values from that directory's module (which is why we added the classes to the biz.__init__.py).

Next, we pass the value of **self.Connection** to the bizobj's creation. Where did the form get a 'Connection' value? Again, this was populated by the DE Wizard: the first step we took in that wizard was to select a connection, and the wizard remembered that and set the form's **CxnName** property (you can check that out in the prop sheet for the form now). When the form is created, it asks the app object for the connection named in the form's CxnName property, and that connection is stored in the form's Connection property. So we pass that connection to the bizobj, which uses it to establish its connection to the database.

So why didn't we see any data? Well, because we never told the bizobj to get any! Let's fix that now: in the code editing window, make sure that the 'Object:' is still set to the form, and find the 'afterInit' method in the 'Method' dropdown. Just add one line after the initial 'def' statement: **`self.requery()`**. After you've done that, run the form again; this time you should see something like:



Date: 09/08/2007
Client: 4
Hours: 3.75
Notes: Delivered a package
Billed? ✓
48 records selected in 0.193 seconds.

Note that there is actual data in the controls, and the status bar at the bottom says something like '48 records selected in 0.193 seconds.'. Congratulations! You have created a form that queried a remote database, grabbed some data, and displayed it in the correct fields - with almost no coding at all! Hey, let's see if there really are a bunch of records there.Close the form, right-click on the big empty slot on the bottom half of the form, and select 'Interactive Controls/ Add Button'. You'll see a button added to that area, but probably down at the very bottom, since buttons don't take up a lot of space. We won't worry about the caption, since we're going to delete the button in a minute, but we want to add a little code to it. Right-click on the button and select 'Edit Code'; the code editing window should come forward, and the onHit() method of the button should already be selected for you. Add the line after the 'def' statement: **`self.Form.next()`**. Run the form again, and see what happens when you click on the button: the various records in the data set are shown in the controls, just as you would expect. But there's something else wrong:: the 'Client' field shows a number, not the client name. The number happens to be the value of the foreign key to the client table for the client associated with that record, but it's not very useful. A user entering in their billable hours will want to pick a name, not an ID number. Let's fix that now.

Populating the Client List

First off, there is a very simple way to solve the display part of things: we could just add a join to the SQL Builder code to pull in the client name column into our data set for hours, and then set the DataField property to that column. But since this is a data entry form, we want to be able to pull in all of the clients so that the user can select the correct one by name, and still have the associated ID value stored in the foreign key field in the hours table. So we'll write a little bit of code to do just this.

There is an excellent explanation of the overall process on the Dabo Wiki: see the page How To Populate And Use List Controls. We'll cover the main points here. The basic approach is to write code in the bizobj to return the names and keys for all the clients, and then use that to populate a list control that the user will select from. So let's open up the ClientBizobj in our favorite text editor, and add the following method:

```
def getNamesAndKeys(self):
    """Return a 2-tuple of lists of the client names
    and their keys.
    """
    crs = self.getTempCursor()
    crs.execute("""select pkid, clientname
            from clients
            order by clientname""")
    ds = crs.getDataSet()
    # Create the lists
    names = [rec["clientname"] for rec in ds]
    keys = [rec["pkid"] for rec in ds]
    return (names, keys)
```

We could have used SQL Builder methods with the temp cursor, but c'mon - how difficult is it to write a simple SQL statement like this? In any case, we run the basic query to get the PKs and names, and create two lists: one containing the names, and the other the PKs.

Next, we make sure that the ClientBizobj is also loaded into the form. So go back to the Class Designer, and open the **createBizobjs()** method in the code editing window. It's easiest to copy/paste the lines that create the hours bizobj, and change the names for the client. It should look like this:

```
def createBizobjs(self):
    hoursBizobj = self.Application.biz.HoursBizobj(self.Connection)
    self.addBizobj(hoursBizobj)
    clientBizobj = self.Application.biz.ClientBizobj(self.Connection)
    self.addBizobj(clientBizobj)
```

Next, go to the design surface and delete the text box for the client. In its place, create a dropdown through the usual right-click/Data Controls/Dropdown List context menu.With the dropdown selected, go to the prop sheet, and set the **DataSource** property to "**hours**", and the **DataField** property to "**clientfk**". We also need to set the **ValueMode** property to "**Key**" to tell Dabo that we don't want to store the displayed text string, but rather the associated Key value. How do we get the displayed strings and keys into the control? Select the control, and set its **RegID** property to '**ClientList**'. As we'll cover later, the **RegID** is a unique identifier within a form that allows for simple, unambiguous object references. Now open the code editing window, select the form in the object dropdown list, and select '**afterInitAll**' from the method dropdown. Add the following code to that method:

```
def afterInitAll(self):
    clientBiz = self.getBizobj("clients")
    names, keys = clientBiz.getNamesAndKeys()
    self.ClientList.Choices = names
    self.ClientList.Keys = keys
    self.ClientList.ValueMode = "Key"

    self.requery()
```

Let's look at what this is doing: the first line asks the form for a reference to the client bizobj; it does this by passing the **DataSource** of the bizobj it is interested to the form's **getBizobj()** method. This is the primary way of getting bizobj references from the form. Next, it calls the method we just created to get the 'names' and 'keys' lists. Next, it sets its **Choices** property to the names, and its **Keys** property to the keys list. For all list controls, the **Choices** property contains the items to be displayed; changing this property changes the options available to the user. The **Keys** property is used "under the hood" to associate the displayed value with a key value; this is done by a 1:1 association between the two: the first Choice entry is associated with the first Key entry; the second Choice with the second Key; etc.

Now since we set the **ValueMode** property to **Key**, the control will display the Choice that corresponds to the linked Key value. Likewise, if the user selects a different Choice, the underlying Key value will be changed.

Finally, we call **self.requery()** to fetch the data for the form, and update all the controls with that data.

We need to add that new method to biz/ClientBizobj.py, so open up that file in a text editor, and add the following method.

```
def getNamesAndKeys(self):
    """Return a 2-tuple of lists of the client names
    and their keys.
    """
    crs = self.getTempCursor()
    crs.execute("""select pkid, clientname
            from clients
            order by clientname""")
    ds = crs.getDataSet()
    # Create the lists
    names = [rec["clientname"] for rec in ds]
    keys = [rec["pkid"] for rec in ds]
    return (names, keys)
```

Ideally, we could just run the modified form, but we have a problem: the client bizobj was loaded before we added the getNamesAndKeys() method, and if we try to run the form now, it won't find that method in the version of the bizobj in memory, and will throw an error. So save everything, exit the Class Designer, and the start it back up with the same command as last time. This time your saved class will open directly. Now you can try running the form, and as you click the button to navigate through the record set, you should see the client name change. Close the running form, and then delete the button; you should see the big empty slot return. We'll fill that next. But first, let's try running this form directly, instead of from the Class Designer. Remember when we created the app with dabo.quickStart(), it set the app's MainFormClass to the default class of dabo.ui.dFormMain? Well, we have a working form now, so let's change main.py to use this form instead. Change your main.py to read:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import dabo
dabo.ui.loadUI("wx")

app = dabo.dApp()
app.MainFormClass = "hours.cdxml"
app.BasePrefKey = "billing_tutorial"
app.start()
```

There are a few changes I made here. First off, I removed the warning comments, as they aren't needed. Second, I set the app's **MainFormClass** to the string "hours.cdxml". Note that .cdxml files are not Python classes; they are just XML files that Dabo knows how to process to create Python classes dynamically at runtime. So instead of setting MainFormClass to an actual Python class as we did before, we set it to the name of a cdxml file that Dabo can turn into a Python class. We could have set it to the full path to the cdxml file, but since we've followed the standard Dabo directory structure created by quickStart(), Dabo will find that file in the 'ui' directory.

The other thing I did is set the app's **BasePrefKey** property. You may have already noticed that Dabo seems to remember some things when you run an app, such as the size and position of windows. This is handled automatically by Dabo's internal preference system; preferences are stored under each application's BasePrefKey to keep them distinct. If not BasePrefKey is set, Dabo uses the path to the app's HomeDirectory. By setting this value here, we can be sure that our app's preferences are kept, no matter if we move the app somewhere else.

Let's test it out: save main.py, and then run it. You should see the form we created in the Class Designer, complete with data. Before closing, resize and reposition the form. Close the form, and then re-run main.py. Note that the form re-opens with the same size and position you left it in.

A Note on Preferences

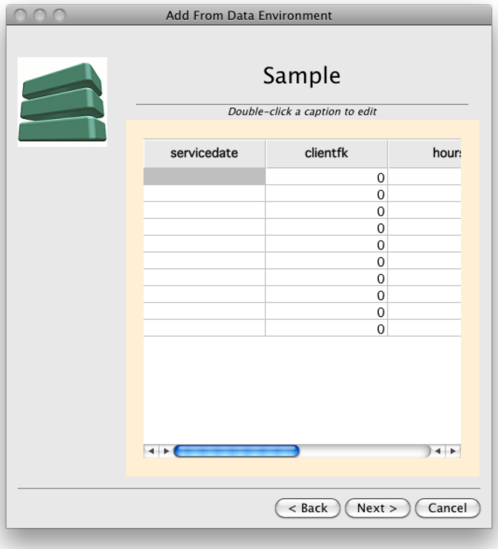
Preferences are one of those things that every application has to handle, so in Dabo we tried to make preference handling as easy as possible. Every object has a property named **PreferenceManager** that refers to the app's preference handling object. To set a preference, simply assign it to an attribute of the PreferenceManager object: **`self.PreferenceManager.meaningOfLife = 42`**. You can represent complex nested preferences using dot notation; just type something like **`self.PreferenceManager.sports.baseball.favorite = "NY Yankees"`**, and Dabo will automatically handle the nested references; *you don't have pre-define the nesting levels*.

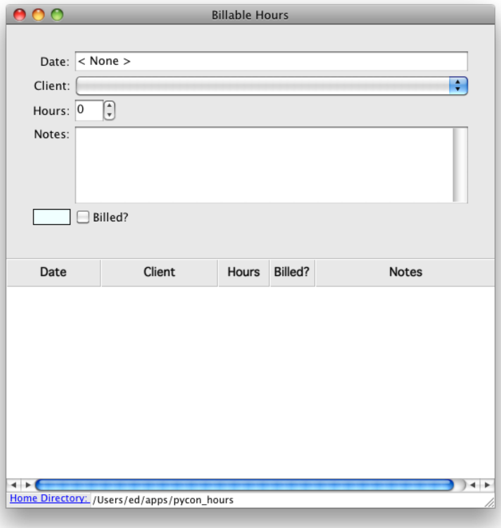
Later on when you want to access a stored preference value, just reference it using the same dot notation name that you created it with: **`return self.PreferenceManager.meaningOfLife`**. Easy, huh?

There is a visual tool for inspecting the preferences database; it's in your 'ide' folder, and is named **PrefEditor.py**. It's not the most polished tool, but it does allow you to edit or delete any of the preferences for your apps.

Creating the Billing Grid

Let's add the grid to the bottom section of the form; we can use the DE wizard again to do that. Re-open the hours.cdxml file in the Class Designer, and right-click on the bottom panel, selecting the DE wizard choice. When you run it, you'll notice that it "knows" that you already have a connection defined for the form, so it uses that and skips the first page. Go through the pages as before, but this time select the Grid layout. When you get to the preview sample page, you should see something like the image below. Like before, you can double-click the header captions to edit them; you can also drag the lines between the columns to change the column widths. When you're done, click 'Next', and this time tell the wizard not to add the bizobj code, since we already have the hours bizobj. When you finish, your form should look like the image below right.

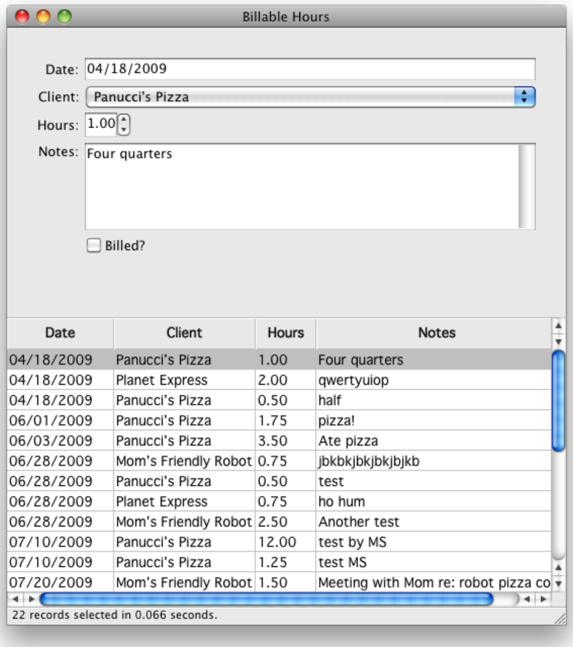




Now fire up the form, and you should see that the grid is populated with all of the existing records from the hours table. There are several things we need to fix with it. First, we have the same problem we had before with the client: the ID and not the name is showing. Since this area isn't data entry, we don't want to use a list control, so we have to add the related table to the SQL for the hours bizobj. The other thing that is noticeable is that we are showing all records, when all we wanted were the unbilled records. We can add that condition to the SQL, too. Finally, the data is unordered, even though it makes more sense to order it by date. So let's add that to the SQL, too. The hours bizobj code should look like this; the lines I just added are marked with arrows. Also note that I had to add the table alias to the calls to **addField()**, since joining with the clients table now introduced an ambiguity, as both tables have a column named 'pkid'. Technically, only that field needs to be fully qualified, but in the interest of consistency, I've changed them all.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import dabo

class HoursBizobj(dabo.biz.dBizobj):
    def afterInit(self):
        self.DataSource = "hours"
        self.KeyField = "pkid"
        self.addFrom("hours")
        self.addJoin("clients",
            "hours.clientfk = clients.pkid")     # <---
        self.addField("hours.clientfk")
        self.addField("hours.hours")
        self.addField("hours.notes")
        self.addField("hours.billed")
        self.addField("hours.servicedate")
        self.addField("hours.pkid")
        self.addField("clients.clientname") # <---
        # Note: if you're using SQLite for your backend,  # <---
        # change the next line to:  # <---
        # self.addWhere("hours.billed = 0")  # <---
        self.addWhere("billed = false") # <---
        self.addOrderBy("servicedate asc")  # <---


    def validateRecord(self):
        """Returning anything other than an empty string from
        this method will prevent the data from being saved.
        """
        ret = ""
        # Add your business rules here.
        return ret
```

Now let's fix the client column of the grid. Select that column in the prop sheet, and change the **DataField** property to 'clientname' instead of 'clientfk'. Save the design, then quit and re-load the design so that the recent changes to the hours bizobj are available. Re-run the form, and you should see the client name in the grid instead of the PK value (below left). The data is ordered correctly, and only unbilled records are in the data set. But notice that the 'Billed?' column shows zeros, not False. That's because MySQL does not have a boolean data type; instead it stores either 1 or 0 as a type called 'tinyint'. We can fix that easily enough by explicitly setting the bizobj's DataStructure property, but think about it: if we are only showing unbilled records, why do we need that column in the first place? Let's delete it - the easiest way is to open the Object Tree, right-click on the 'billed' column, and select 'Delete'. Now when we run the form it will look like this:



Date: 04/18/2009
Client: Panucci's Pizza
Hours: 1.00
Notes: Four quarters
☐ Billed?

Date	Client	Hours	Notes
04/18/2009	Panucci's Pizza	1.00	Four quarters
04/18/2009	Planet Express	2.00	qwertyuiop
04/18/2009	Panucci's Pizza	0.50	half
06/01/2009	Panucci's Pizza	1.75	pizza!
06/03/2009	Panucci's Pizza	3.50	Ate pizza
06/28/2009	Mom's Friendly Robot	0.75	jbkbkjbkjbkjbjkb
06/28/2009	Panucci's Pizza	0.50	test
06/28/2009	Planet Express	0.75	ho hum
06/28/2009	Mom's Friendly Robot	2.50	Another test
07/10/2009	Panucci's Pizza	12.00	test by MS
07/10/2009	Panucci's Pizza	1.25	test MS
07/20/2009	Mom's Friendly Robot	1.50	Meeting with Mom re: robot pizza co

22 records selected in 0.066 seconds.

Making This a Data Entry Form

Right now the form as it exists is great for displaying data that's already been entered, but we said at the beginning that we wanted to make this a data entry form. That means that when we run the form, we are interested in adding new records, not just viewing old records. So one of the things we'll need to do is call the form's **new()** method as soon as the form is created. The place to do that is the form's **afterInitAll()** method, so we're going to add the call to **self.new**() to the end of that method. If you run the form now, you will probably get some error messages, because new records have empty values, and so the **clientfk** field, which is an integer, will get a value of zero, which is not present in the Keys property of the client dropdown. We'll also get an "empty" date, which is probably not what we want.

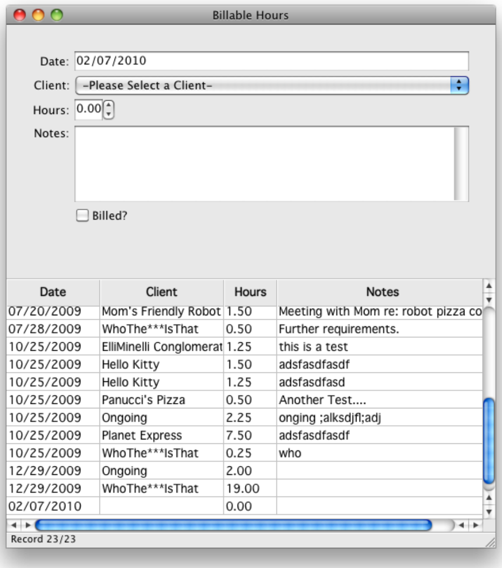
To solve these problems, we're going to have to make a few tweaks. First, let's add an option to the client dropdown for an unspecified client. In the code editing window, select the form object, and modify the code to insert the 'unspecified client' option.

```
def afterInitAll(self):
    clientBiz = self.getBizobj("clients")
    names, keys = clientBiz.getNamesAndKeys()
    names.insert(0, "-Please Select a Client-")
    keys.insert(0, 0)
    self.ClientList.Choices = names
    self.ClientList.Keys = keys

    self.requery()
    self.new()
```

Save these changes, and exit the Class Designer. For the date, most of the time we'll want it to be today's date. Fortunately, Dabo provides a simple way to provide default values for new records: bizobjs have a **DefaultValues** property, which is a dict with column names as the keys and the defaults as the values. So open up HoursBizobj.py in your text editor, and add the following line:

```
self.DefaultValues = {"servicedate": datetime.date.today()}
```

Of course, make sure that you add the `import datetime` statement to the top of the file. When you've saved those changes, run **python main.py** from the terminal to see how the app is working. If you did everything right, you should see a form like this, but with today's date:



Date: 02/07/2010
Client: –Please Select a Client–
Hours: 0.00
Notes:
☐ Billed?

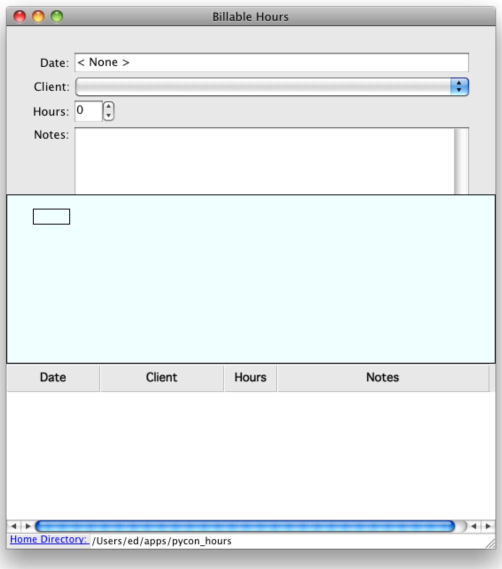
Date	Client	Hours	Notes
07/20/2009	Mom's Friendly Robot	1.50	Meeting with Mom re: robot pizza co
07/28/2009	WhoThe***IsThat	0.50	Further requirements.
10/25/2009	ElliMinelli Conglomerat	1.25	this is a test
10/25/2009	Hello Kitty	1.50	adsfasdfasdf
10/25/2009	Hello Kitty	1.25	adsfasdfasd
10/25/2009	Panucci's Pizza	0.50	Another Test....
10/25/2009	Ongoing	2.25	onging ;alksdjfl;adj
10/25/2009	Planet Express	7.50	adsfasdfasdf
10/25/2009	WhoThe***IsThat	0.25	who
12/29/2009	Ongoing	2.00	
12/29/2009	WhoThe***IsThat	19.00	
02/07/2010		0.00	

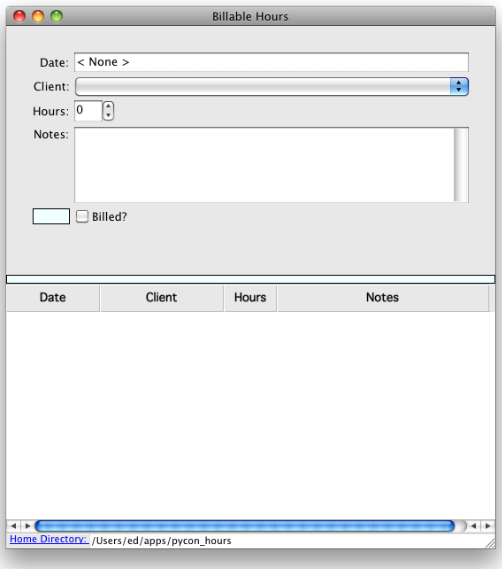
Record 23/23

If we play around with this, another problem becomes obvious: if we click the up or down arrows of the spinner, the value increases by an hour at a time, while we will be billing in quarter-hour increments. To fix this, open the form in the Class Designer, and select the spinner. In the prop sheet, change its **Increment** property to '0.25'. Now when you click the arrows, the value will change by quarter-hour increments. And while we're here, let's change the **Max** property to 24; presumably we won't be billing more than 24 hours a day!

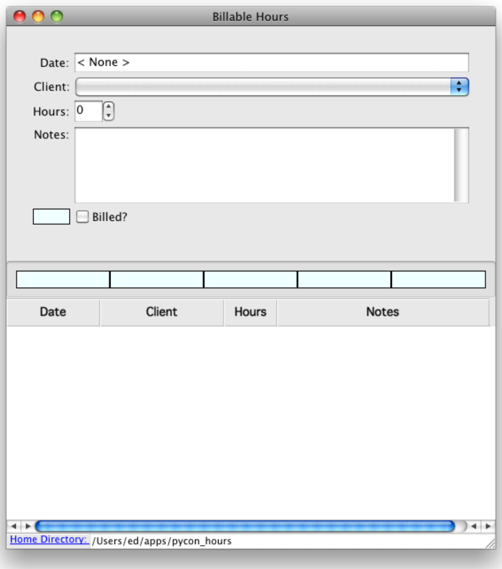
Saving Your New Record

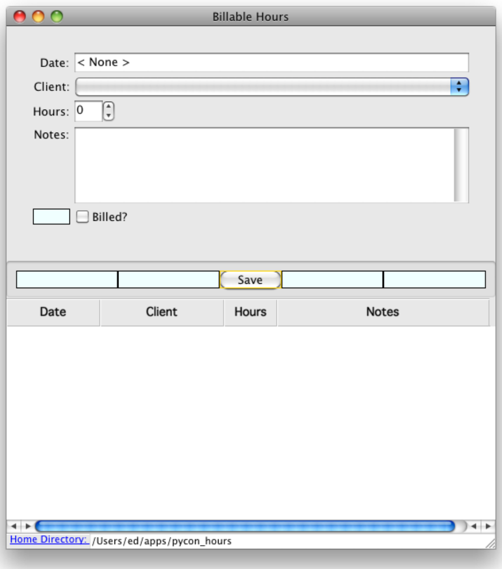
We haven't yet provided a way for the user to tell the form that the new record's data is complete, and to save it to the database. We're going to need a new "slot" to add some buttons, so open up the form in the Class Designer, and go to the Object Tree. Find the grid (not the Grid Sizer), and right-click on its node. Select 'Add Slot Above' from the context menu. You'll see a new rectangular section appear (below left) between the controls at the top and the grid at the bottom; it will probably overlap the controls in the top section, but don't worry about that now - we're going to fix it. Right click on this new slot, and select 'Edit Sizer Settings'. Change the proportion to 0, and click 'OK'. The slot will now be a skinny little strip (below right).





We're going to add some buttons to that strip; the buttons will be arranged horizontally, so that means that we need a horizontal sizer. So right-click on the skinny strip, select 'Sizers / Add New Horizontal Sizer', and set the number of slots to 5. Just for fun, check the 'Add Sizer Box?' option. You could add a caption for the box, but we really don't need one, so leave that blank. The form will now look like the image below left. Let's add a button to the middle slot in that strip. You should know the drill by now. Change the button's Caption to 'Save'. The form should now look like the image below right.

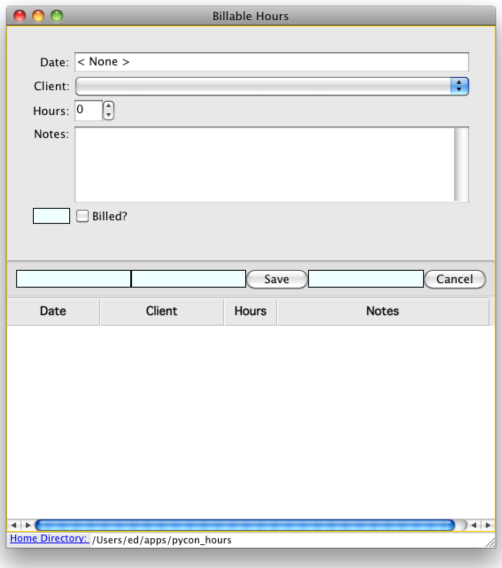


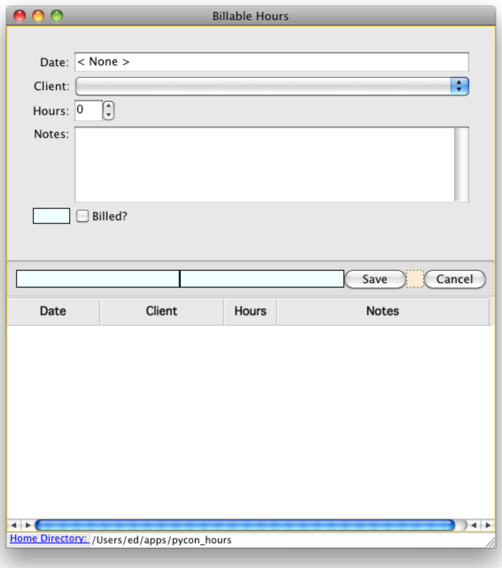


Right-click on the button, and select 'Edit Code'. The code editing window should come forward, with the button as the selected object. It even selected the onHit method, since that's the most likely method that you'll need to work with when dealing with buttons. Add the line **`self.Form.save()`** after the 'def' line. That's it - all you need to do to save all your changes throughout the form's controls is to make that one simple call.

We're going to need a Cancel button, too, but instead of adding it as we did the Save button, let's simply copy and paste the button! Right-click on the button, and select 'Copy'. Now right-click on the rightmost of the new slots - you'll see a 'Paste' option at the top of the context menu. Select that, and a copy of the save button appears in that slot. All we have to do is change its Caption to 'Cancel', and edit its onHit method to say **`self.Form.cancel()`** instead.

The buttons look OK, but there's too much space between them (below left). We could delete the slot between them, but then they'd be jammed against each other. Instead, let's add a Spacer between them. From the right-click/Sizers menu, select 'Add New Spacer'. Change the width to 20, and the form should now look like the image below right.





Save this and exit. We could run the app, but before we do, let's talk about data validation. After all, that's one of the most important functions of a bizobj, but we really haven't discussed it at all. The changes that we've made present a perfect validation opportunity: if they don't pick a client from the dropdown, we obviously can't bill anyone! So let's add a validation check for that.

Open HoursBizobj.py in your text editor, and go to the **validateRecord()** method. Modify it so that it reads:

```
def validateRecord(self):
    """Returning anything other than an empty string from
    this method will prevent the data from being saved.
    """
    ret = ""
    # Add your business rules here.
    if self.Record.clientfk == 0:
        ret = "You must select a client"
    return ret
```

What this is doing is checking for a value of zero for the client FK. If it finds such a value, it returns the error message text. The framework will interpret any non-empty return value from validateRecord() as indicating a validation failure, and will raise a BusinessRuleViolation exception.

Save your bizobj changes, and run main.py. Add some values to the form, but leave the client dropdown at the default, unspecified selection. Now click the save button; you should get a dialog that displays the error message you returned from the validateRecord() method.

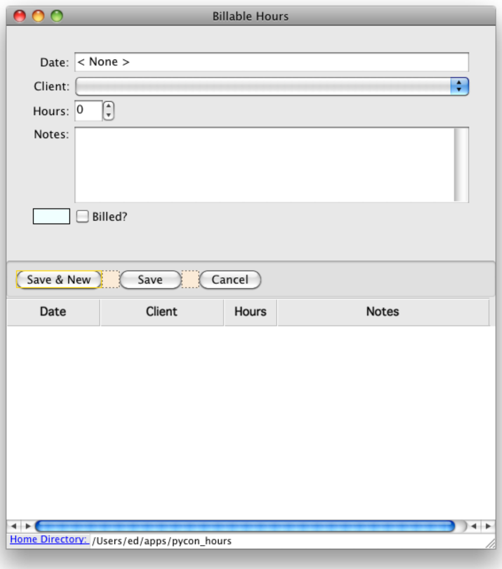
Fix this validation problem by selecting a proper client from the dropdown, and then click 'Save' again. This time you should not get the error message; instead, you should see a confirmation message in the form's status bar.

Next Steps

There are lots of things we can do to make this application more functional, but you can see that we've gotten pretty close to what we need fairly quickly, without any cryptic coding efforts. One thing that would be simple to add is to have the app automatically call **`new()`** after a successful **`save()`**. It would be nice to be able to enter multiple billable sessions in one sitting. So let's add a button that will do that.

Open the form in the Class Designer, and copy the Save button. Paste it in the leftmost slot. Change the Caption property to '**`Save && New`**'. Why the double ampersand? Well, this is a quirk of the wxPython toolkit: it interprets a single ampersand in a Caption as a symbol that will make the following character the button's hotkey. We actually want to see an ampersand in the Caption, so we need to double it up.

Now copy the spacer we created, and paste that in the last remaining slot. Hmmm... this isn't what we wanted (below left). We want the buttons right-aligned, but how do we do that? Remember, the buttons are in a horizontal sizer, and this horizontal sizer is in a slot of the main vertical sizer. Since the horizontal sizer is not wide enough to take up the whole slot, its Expand setting is filling up the rest. So let's fix it! Go to the Object Tree and locate it: it should say 'BorderSizer: Horizontal'. Right-click to get the 'Edit Sizer Settings' dialog. Whoa! This looks very different! (below right).

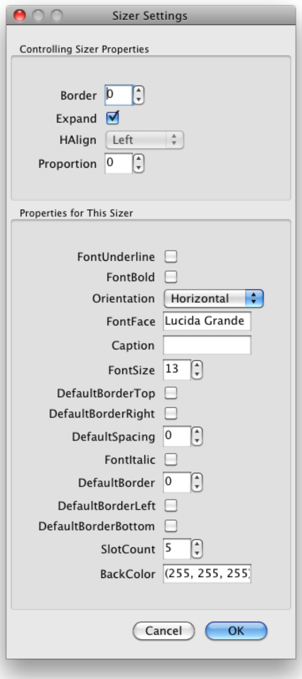


Billable Hours

Date:	< None >
Client:	
Hours:	0
Notes:	
☐ Billed?	

Date	Client	Hours	Notes

Home Directory: /Users/ed/apps/pycon_hours



Sizer Settings

Controlling Sizer Properties

Border	0
Expand	✓
HAlign	Left
Proportion	0

Properties for This Sizer

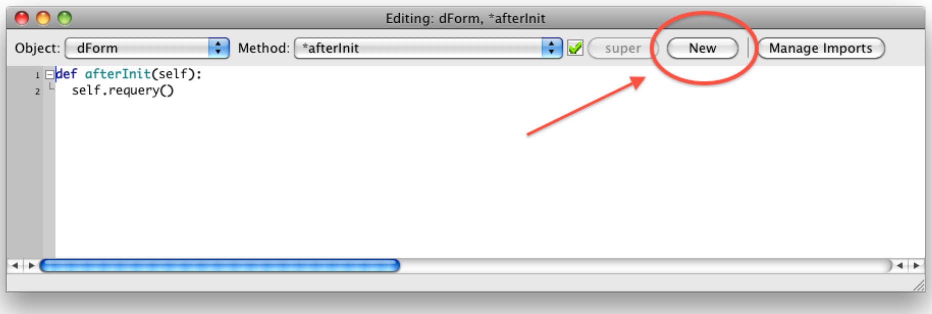
FontUnderline	☐
FontBold	☐
Orientation	Horizontal
FontFace	Lucida Grande
Caption	
FontSize	13
DefaultBorderTop	☐
DefaultBorderRight	☐
DefaultSpacing	0
FontItalic	☐
DefaultBorder	0
DefaultBorderLeft	☐
DefaultBorderBottom	☐
SlotCount	5
BackColor	(255, 255, 255

This is probably where most people get *really* confused when working with sizers. Since sizers can be nested inside of other sizers, there are settings that affect how the outside sizer treats them (top section), and also how this sizer treats its members (bottom section). We have no issue with the latter; it's just useful to know that you can change any of these settings all from a single location.

To fix our current problem, in the top section uncheck the 'Expand' setting, and change the HAlign setting to 'Right'. Click 'OK', and the form should look the way we want it.

Now what about the code? We could add it to the **`onHit()`** method of our new button, but there's a general design guideline that goes with the Chain of Responsibility Pattern we mentioned earlier: objects should only know as little as possible to do their job. So yeah, we could put a bunch of code in the button, but controls are supposed to be pretty dumb. They should simply call a method of their Form to let the form know that the user wanted to do something, and that's about it. In this design, the Form is the central brains of the operation. So just add a single line to the button's onHit() method: **`self.Form.saveAndNew()`**.

There's just one problem: the form doesn't have a method by that name! Actually, it's not a problem at all, as we can add it very easily. Go to the Code Editing window, and select the form in the object dropdown. The Method dropdown contains the names of the most common methods and available event handlers for the selected object, but if you need to create a new method, no problem – that's what the 'New' button is for!



```
def afterInit(self):
    self.requery()
```

Click that button, and enter the name of your new method; in this case, it's 'saveAndNew'. You'll see that method appear in the Code Editing window, and we can add our code right away:

```
def saveAndNew(self):
    if self.save():
        # A failed save will return False. We only want to proceed if
        # the save succeeds.
        self.new()
        self.serviceDate.setFocus()
```

Wait a second - what's this 'self.serviceDate' in the last line? It seems like it should be a reference to the date control, since it would make sense to have the form set focus to that when entering a new record, but we didn't create any such reference! That's true, but it's pretty easy to do so, by using the **RegID** property of all Dabo controls. This is a property that is generally empty, but if you set it, it *must* be unique across all controls on any given form, and once it's set, it cannot be changed. This ensures that it will always be possible to refer to that object unambiguously. You could also use a dot-separated list of the names in the containership hierarchy of the control (e.g.: self.panel1.pageframe1.page3.panel2.button1), but that's very fragile, as controls are frequently moved and/or renamed during development.

When a control with a non-empty RegID is instantiated, it "registers" itself with its form (that's the 'reg' part of the 'RegID' name), and that RegID is added to the form's namespace. From that point on, any control can reference that registered control by simply using **self.Form.*RegID***, where *RegID* is the RegID of the control.

Let's set that RegID for the date control. Select it in the design surface, and go to the prop sheet. Find the RegID property, and set it to 'serviceDate'. That's it! Try it out: save the design and run your app. Add a billable record, and then click the Save & New button. The record should be saved and the controls should blank out for a new record. Focus should also be on the date control.

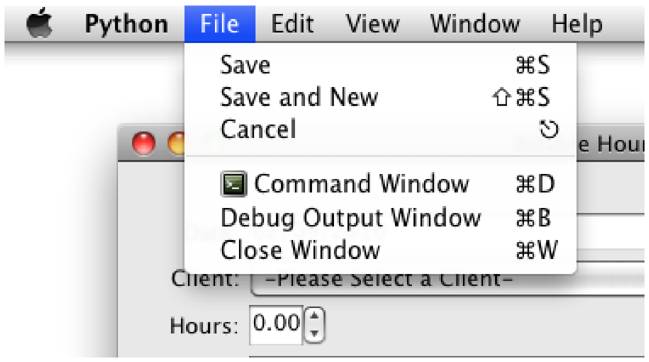
Working with Menus

We haven't really touched on menus in this tutorial, because the visual tool we started to create for menus never really got the attention it needed, and basically doesn't work. So all menu creation is done in code. But it's really not that difficult, and to demonstrate, let's add menu options for the

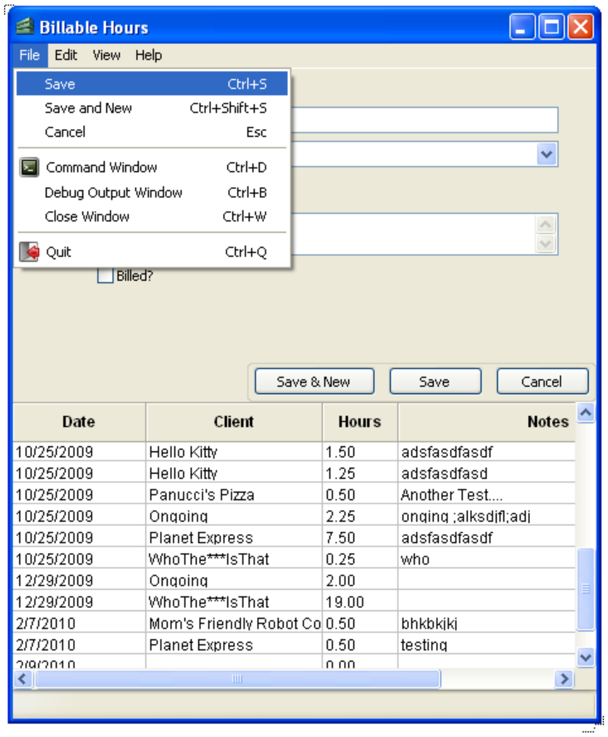
'Save & New' button we added to the form. Here's the code; we'll add it to the form's afterInit() method.

```
def afterInit(self):
    mb = self.MenuBar
    fm = mb.getMenu("File")
    fm.prependSeparator()
    fm.prepend("Cancel", HotKey="Esc", OnHit=self.onCancel)
    fm.prepend("Save and New", HotKey="Ctrl+Shift+S",
            OnHit=self.saveAndNew)
    fm.prepend("Save", HotKey="Ctrl+S", OnHit=self.onSave)
    self.requery()
```

Each form has its own menu bar, with a reference to it in its **MenuBar** property. You can get references to each menu in the menu bar by passing that menu's Caption to the menu bar's **getMenu()** method. We want these menu items to appear above the default items, so we'll use the **prepend()** method and add them from the bottom up. First we add a separator, then the items for the button functions. Note that we can pass in the **HotKey** key combination as well as the handler for that menu (using the standard **OnHit** parameter). The form class comes with handlers for onSave() and onCancel(), but the saveAndNew() method we added earlier is not an event handler (i.e., it doesn't accept an event object parameter). That's easy enough to fix: we can just add an optional **evt=None** parameter to that method's signature. Save, quit, and run the app. You should now see our custom menu:



Note how the cross-platform stuff is handled for you: I specified **Ctrl-S** as the shortcut, and Dabo knew it was running on a Mac, and substituted the Command Key symbol; it also showed the symbol for Escape. If you were to run the app under a different platform, you wouldn't need to do anything different at all. Here's a shot of the same app running unchanged under Windows XP:



Date	Client	Hours	Notes
10/25/2009	Hello Kitty	1.50	adsfasdfasdf
10/25/2009	Hello Kitty	1.25	adsfasdfasd
10/25/2009	Panucci's Pizza	0.50	Another Test....
10/25/2009	Ongoing	2.25	onging ;alksdjfl;adj
10/25/2009	Planet Express	7.50	adsfasdfasdf
10/25/2009	WhoThe***IsThat	0.25	who
12/29/2009	Ongoing	2.00	
12/29/2009	WhoThe***IsThat	19.00	
2/7/2010	Mom's Friendly Robot Co	0.50	bhkbkjkj
2/7/2010	Planet Express	0.50	testing
2/9/2010		0.00	

Adding a Report

What's a billing application if you can't generate reports? Let's add the most important report in a consultant's life: the client invoice. We'll add another menu item that instantiates another form, and that form will take some user input to generate the report cursor. We'll use the ReportDesigner to design the report.

(The rest of this section is included in the online version of the tutorial notes).

Running Your Application Over the Web

Web applications are great because they require zero deployment and zero upgrade pain. Just stick your code on a webserver, and by navigating to your app by putting your URL in a web browser, anyone can run your app. The big downside, though, is that while you can use Python for some of your logic, all of your user interface has to be in HTML and Javascript.

Dabo has an exciting new technology called **Springboard**. Think of it as being analogous to a web browser: you launch Springboard, enter a URL, and you have a Dabo application running on your desktop. No installing anything; Springboard handles all of that for you. Even better: update the source code on your server, and Springboard will grab the latest version every time, so you enjoy the same zero deployment and zero upgrade pain as browser-based web apps.

There are other advantages to running Dabo apps over the web, with security being at the top of the list. For the sort of apps we've developed above, you have to serve your data across the web by opening up your database to external requests. While you can certainly take steps to make that as secure as possible, it's still another entry point for attackers to compromise your site. Also, your business logic may contain proprietary information about how your business works that a competitor might be able to use against you; with a Springboard-based app, business logic remains on the server; only the UI and proxy business objects are sent to the client. These local business objects know how to communicate with the server, but do not do any data access or validation.

Setting Up the Server

There is an excellent document that describes this process that we've already published: Creating a Web Application with Dabo. You can refer to that for details about the process, but it comes down to a few basic steps:

- Set up the web server (the examples all use Pylons)
- Copy your app source code to the server
- Add the apropriate SourceURL to your app
- Create the RemoteBizobj classes
  - Define the database connection for the RemoteBizobj classes
  - Move any business logic from your app code to the RemoteBizobj classes.
-
- Define the mapping to the RemoteBizobj classes in the controller

For this we'll skip the first step, as we already have a server set up: **http://daboserver.com**, so later we'll add that to the app as the **SourceURL**. Next, we need to copy the source code over to the server. Once it's there, all of the rest of the instructions assume that you're operating on the server, not your local machine.

Open main.py for your app, and add the line to set the **SourceURL**. While we're at it, let's pass all of these to the app's initial creation. The effect is the same; this is more of a personal preference.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import dabo
dabo.ui.loadUI("wx")

app = dabo.dApp(MainFormClass="hours.cdxml",
        BasePrefKey="billing_tutorial",
        SourceURL="http://daboserver.com")
app.start()
```

Now remember, even though we're editing code on the server, this code will be copied to the user's local machine by Springboard, and run there. This may be a bit confusing, so if it helps, remember that nothing in the source directory on the server is ever executed on the server: they are just files to be served up.

Next, we need to create the bizobj on the server to handle requests from the client. The **daboserver** Pylons app handles the actual HTTP layer, and will call the **bizservers.py** controller. This controller will then map the request's DataSource to our RemoteBizobj class to be handled; we'll create the RemoteBizobj class file in the *controllers* directory, so that the classes will be available to the bizservers controller. We'll explain the mapping in a little bit. But first, let's look at this 'RemoteBizobj': how is it different? Well, let's start by showing the completed RemoteBizobj for the *hours* table:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import dabo

class HoursBizobj(dabo.biz.RemoteBizobj):
    def defineConnection(self):
        self.setConnectionParams(
                dbType="MySQL",
                host="dabodev.com",
                database="pycon",
                user="pycon",
                plainTextPassword="atlanta")

    def validateRecord(self):
        """Returning anything other than an empty string from
        this method will prevent the data from being saved.
        """
        ret = ""
        # Add your business rules here.
        if self.Record.clientfk == 0:
            return "You must select a client"
        return ret
```

There are a couple of things to note. First, we inherit from **dabo.biz.RemoteBizobj**, not **dabo.biz.dBizobj** - this gets us some additional web smarts that are not in the regular bizobj class. Also note the **defineConnection()** method: with regular bizobjs, the app handled the connections, and passed it to the bizobj when it was created. For remote bizobjs, we have to put the connection info in the bizobj, so this is how it's done. Finally, we have the **validateRecord()** method that we simply copied from the local version. We can delete this code from the local app code, since it will never be called; all validation happens on the server.

Now all we really have left to do is to define the mapping between the DataSource and the RemoteBizobjs. To do this, open up **bizservers.py** on the server in a text editor, and scroll down until you see the comment section labeled 'START OF CUSTOMIZATION'.

```
#-------------------------------------------------------
#           START OF CUSTOMIZATION
#-------------------------------------------------------
#       The next two sections are the only parts you have to edit
#       for your application. Be sure to specify the bizobj classes
#       you will be using, and make sure that they are located in
#       the same 'controllers' directory as this file.
#-------------------------------------------------------
# Import the bizobj classes here that will be used in this application
# Then be sure to link them to their DataSource in the 'bizDict'
# definition below.
## NOTE: the next line is an example from the demonstration app.
## Be sure to CHANGE it to whatever is required for your app.
from PeopleBizobj import PeopleBizobj
from ActivitiesBizobj import ActivitiesBizobj
from HoursBizobj import HoursBizobj
#-------------------------------------------------------
# The bizobj class *MUST* be defined here for each data source that is
to be
# handled by this server. Be sure that these classes are imported
above.
## NOTE: as mentioned above, this is for the demo app.
dabo._bizDict = {
      "people": PeopleBizobj,
      "activities": ActivitiesBizobj
      "hours": HoursBizobj}

# The path to the server copy of the web application source files
*MUST* be
# defined here. It is used to compare local app manifests in order to
# determine what changes, if any, have been made to the app.
sourcePath = "/home/dabo/appSource"
#-------------------------------------------------------
#           END OF CUSTOMIZIATION
#-------------------------------------------------------
```

This code is copied from the Springboard server on daboserver.com, which was already set up to handle an app that used two bizobjs: 'PeopleBizobj' and 'ActivitiesBizobj'. I've added code to two places: first, I've added the **import** statement; remember that we created the HoursBizobj.py file in the web server's *controllers* directory, so it's available to be imported here. Second, I've added the key 'hours' to the **_bizDict** attribute of the controller, and its value is the RemoteBizobj for the hours table.

Wait a second – where does the 'hours' DataSource come from? Well, that's part of the "glue" built into bizobjs: when they are in an app with its **SourceURL** set, they "know" how to send requests over the web to their RemoteBizobj counterpart. Each of these requests passes the local bizobj's DataSource as part of the URL; the Pylons controller parses out the DataSource from the URL, and uses its **_bizDict** mapping to create an instance of the appropriate RemoteBizobj class to handle the request. But the real beauty of this is that you don't really need to do very much to take a locally-installed desktop app and turn it into an app that is run by Springboard over the web.

(The rest of this section is included in the online version of the tutorial notes).

Contact Information

We are going to be adding to this tutorial document all the way up to (and possibly after!) PyCon, even though we had to submit this to the printer beforehand. The latest version will be available online at http://dabodev.com/pycon_tutorial, so be sure to visit that link to get the latest and greatest!

For questions about developing applications in Dabo, subscribe to the dabo-users email list:
http://leafe.com/mailman/listinfo/dabo-users
There are many helpful people on that list who can assist you with your problems.

You can always contact the authors of Dabo:

Paul McNett: p@ulmcnett.com
Ed Leafe: ed@leafe.com