DESIGN PATTERNS FOR TEST DRIVEN DEVELOPMENT IN ACCESS VBA

Todd Schultz, PhD, tschultz@aug.edu

ABSTRACT

Application of appropriate design patterns can extend test driven development within an Access Visual Basic for Applications environment to allow unit testing of object class properties and methods. While this approach is not as simple as automating testing of functions in modules, the approach is tractable for intermediate-level developers and leads to significant benefits derived from a test driven approach.

INTRODUCTION

Test driven development (TDD) is a notable shift in attitude toward the routine activities of developers. Under TDD code is developed in small nuggets to resolve a particular test case (without breaking those which have already been passed); this is in contrast to longer periods of development to meet broader sets of functions. This forces developers to make many small 'releases' which allows focus on small bits of functionality coupled with routine testing of all functionality developed to date. This shift can be attributed to hardware improvements (which make running thousands of tests after each code change) as well as a maturing of software engineering away from ad hoc solutions toward software patterns.

As strong as TDD is as a metaphor for creating software, it is a challenge to establish a TDD discipline, especially to those new to programming. TDD often requires advanced knowledge of coding technique and demands a mature understanding of how software solutions get deployed. Add to this the complexity of using testing suite software across an organization and TDD can easily become one of those good habits we do not build in ourselves and others.

Test driven development is most often associated with pure object oriented development [6] affecting attitudes toward TDD and programming [8]. Visual Basic for Applications (or VBA) is an incredibly available programming language (e.g., it is embedded in every instance of Microsoft Word, Excel, PowerPoint, etc.), but this availability and ease of use cause it to not be taken seriously by academics and purists [10] and there is interest in eliminating some of the implementation burdens associated with TDD [5]. As with most developer tools, the problem is not primarily with the language but the discipline and standards with which it is applied. In addition to providing a good introductory vehicle for programming principles and platform for applications within the enterprise, VBA can be a suitable framework for test driven development. Further, a test driven environment can easily be built upon Access’s strong rapid-application-development (RAD) query and reporting tools.

For those interested in applying these ideas in the classroom, lecture notes, complete code, and student assignments are available on our e-learning web site at {identification omitted for review}. The example we highlight below is designed to be easily extended by students to develop similar, parallel functionality and the assignments are in the context of a realistic textbook case.

EXAMPLE PROBLEM CONTEXT

Applying the principle that it always helps to see an example, our development surrounds a small database to support an antiques mall (see [2] running Case 1) where booths for stores are rented to dealers. Among the tables in this project is a Booths table with the definition shown in Figure 1.

![Booths Table Schema](image_url)

Figure 1 – Booths Table Schema
In an earlier paper [9] we described how a TDD approach could be applied without any coding to the problem of developing a function which calculates the area of a booth from the Size field which is text of the form “L x W” where L and W are the length and width of the booth. We include a brief review of that content in the following section since it sets the tone for the current work.

It is a premise of TDD that developers should write more code testing than they should to solve the problem. With Access and using the embedded VBA language, it is possible to develop test suites with minimal coding, even when testing more advanced architectures like class properties and methods. Functions residing in modules can be used directly in the Jet database engine while Access is running (i.e., then can be called from queries, forms, and reports). Module subroutines and class properties and methods cannot be used as directly but, as described after the following section, use of the bridge and façade patterns [4] combines direct data engine access to components with an object oriented design.

**NO CODING TDD IN ACCESS VBA**

Functionally, calculating a booth’s area obviously requires parsing out the length and the width from the Size (a string type), then multiplying these two quantities. We specified that BoothArea should return a VBA integer (which has a maximum of 32,767 and is more often today called a short (integer) in programming languages) and that a zero area should be returned if any ‘issues’ arise: malformed Size parameter, negative L or W, overflow, etc. One might quibble with these particular choices in the specification, but they establish the baseline for the development and, more importantly, the testing.

Test driven development is based on two principles: write unit test(s) first and only write (or modify) code when a test fails [1] and [7]. Unit tests are based upon assertions of what result is expected based upon a known set of inputs. Within Access, assertions (for functions within a module) are easily created by building a table; something like the following:

<table>
<thead>
<tr>
<th>BoothTestPK</th>
<th>TestDescription</th>
<th>BoothSize</th>
<th>expBoothArea</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Normal case - 2 digits w/spaces</td>
<td>18 X 12</td>
<td>216</td>
</tr>
<tr>
<td>2</td>
<td>Normal case - 1 digit x 3 digits w/spaces</td>
<td>5 X 125</td>
<td>625</td>
</tr>
<tr>
<td>3</td>
<td>Normal case - 4 digit x 1 digits w/spaces</td>
<td>1305 X 5</td>
<td>6525</td>
</tr>
<tr>
<td>4</td>
<td>Normal case - 2 digits no spaces; lc x</td>
<td>18x12</td>
<td>216</td>
</tr>
<tr>
<td>5</td>
<td>Normal case - 1 digit x 3 digits varying spaces</td>
<td>5X 125</td>
<td>625</td>
</tr>
<tr>
<td>6</td>
<td>Normal case - 4 digit x 1 digits w varying spaces</td>
<td>1305X 5</td>
<td>6525</td>
</tr>
<tr>
<td>7</td>
<td>Area over flow (not perimeter)</td>
<td>16000x5</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Area and perimeter overflow</td>
<td>20000x20000</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>No X char</td>
<td>18 z 32</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>No X char, extra digits</td>
<td>144 z 35</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>Junk chars</td>
<td>bad text</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>Numbers &amp; spaces only</td>
<td>144 33 22</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>Numbers only</td>
<td>323513</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3 – Sample Unit Test Assertions

Beyond an autonumber primary key and a text description of the test, the table simply contains a list of candidate booth sizes (input argument) and the expected booth areas (function result). An important conceptual principle is that this table would be created before any coding was done. From a TDD perspective, this list is the specification for the function (or at least a good part of the spec).

Executing the test suite is easy given the Access and VBA development environment. Write a query that uses this table as input, calculate the booth area, and compare that with the expected booth area; something like where the key column definitions are

calcBoothArea: BoothArea([BoothSize])
TestFailed: (expBoothArea)<>[calcBoothArea]
The test suite is run by opening the query in datasheet view or using the query as input to a report or form which prints or displays the tests (all, only passed tests, or only failed tests) in a nicer format. TestFailed is a Boolean variable indicating the result of the test.

Following a pure TDD approach, execution of the test suite will fail at its first trial since the BoothArea function hasn’t been written. Writing at least a BoothArea function stub (which returns zero regardless of the Size argument) results in several failed tests (though tests 7 through 13 from Figure 3 would actually pass with this BoothArea function so in a sense, this nascent function is ‘half right’). At this point we begin in earnest the coding we all know and love and it does not end until all the tests pass. Once the tests pass, the code is refactored and additional tests are created in attempts to break the code or in response to bugs which have been uncovered.

EXTENSION TO MORE COMPLEX ARCHITECTURES

This simple and relatively obvious approach to ‘no code testing’ relies heavily upon the fact that the VBA runtime is brought into the Jet database query engine while the user is in Access; if this query were executed outside Access by establishing an OLEDB connection and creating a command to the query, the BoothArea function would not be available. Of course the testing query was written with full knowledge of how the component being tested worked and was a form of coding. Automated testing does not mean that somehow the code will be automatically tested without someone intelligently establishing the test suite; instead, quick and clever test definitions which are easily run can automatically and reliably unit test the code.

The availability of a full programming component within a database engine is not a particular property of Jet in Access, nor databases in general: one of the controversial features of Microsoft’s release of SQL Server 2005 was that it brought the .Net runtime inside the database engine. This raises valid concerns about developers undoing the inherently efficient database processing with their own inefficient code. For our purposes, though, it allows us to leverage powerful database processing against fairly complex code structures using well established design patterns.

Continuing the explanatory example, suppose the rental booth requirements were met in code by establishing a Booth object with a persistence layer in the table schema shown in Figure 1. Business rules for this application ‘thought’ about rent per square foot, not per booth as defined by the field in the table. Presumably there are other legacy applications using this table which expect the Rent field to be for the booth so our business object provides a façade for the persistence object by making it appear that RentPerSqFt is a ‘natural’ part of the base object.

To that end, the following (see Figure 5) class Booth was begun by the VBA developer (note that the object encapsulates the fact that rent is stored per booth via the public property RentPerSqFt):

Option Compare Database
Option Explicit
Private mBoothID As String
Private mSize As String
' mRent is rent for Booth, not per Sq Ft
Private mRent As Currency

Public Property Get BoothID() As String
    BoothID = mBoothID
End Property

Public Property Get Size() As String
    Size = mSize
End Property

Public Property Let Size(BoothSize As String)
    mSize = BoothSize
End Property

Public Property Get Area() As Integer
    {Code hidden to save space}
End Property

Public Property Get RentPerSqFt() As Currency
    {Code hidden to save space}
End Property

Public Property Let RentPerSqFt(Amount As Currency)
    mRent = Amount * CCur(Me.Area)
End Property

Public Sub Init(BoothID As String, BoothSize As String, RentPerSqFt As Currency)
    {Code hidden to save space}
End Sub

Public Sub DBCreate(BoothID As String, BoothSize As String, RentPerSqFt As Currency)
    Call Me.Init(BoothID, BoothSize, RentPerSqFt)
    Call Me.DBUpdate
End Sub

Public Sub DBRead(BoothID As String)
    ' Todo: Develop code to retrieve Booth
End Sub

Public Sub DBUpdate()
    ' Todo: Develop code to save Booth
End Sub

Public Sub DBDelete()
    ' Todo: Develop code to delete booth
End Sub

Figure 5 – Booth Class (in progress)

Before doing any more Booth coding (and perhaps even before getting this far), unit tests for the Booth properties and methods should be developed. This is where the bridge pattern can be applied by building a testing function which instantiates a Booth object and returns properties as test results. Not a difficult concept but another layer of abstraction. For instance, the following module function tstSetGetRentPerSqFt allows us a handle on the Booth class properties which set and get the rent per square foot (it is a fact of TDD that testing component names often get wordy):

Function tstSetGetRentPerSqFt(Size As String, RentPerSqFt As Currency) As Currency
    Dim tstBooth As Booth
    Set tstBooth = New Booth
    tstBooth.Size = Size
    tstBooth.RentPerSqFt = RentPerSqFt
    tstSetGetRentPerSqFt = tstBooth.RentPerSqFt
End Function

Figure 6 – Bridge Function Class Properties

This function is then wrapped in a query which calls it based on a table which contains the data for the individual tests; this is similar to the process outlined in the previous ‘No Coding …’ section. Results of the test query (with only two tests) are shown below in Figure 7. Note the standard columns for TestSet, TestFailed, TestPK, and TestAddedDT (date and time test was added); this allows for systematic tracking of tests and results. Also the third test failed since the expectation when setting with a negative rent would be a zero rent.

<table>
<thead>
<tr>
<th>TestSet</th>
<th>TestFailed</th>
<th>TestPK</th>
<th>TestAddedDT</th>
<th>Test Description</th>
<th>Size</th>
<th>RentPer SqFt</th>
<th>ExpRent PerSqFt</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetRentPerSqFt</td>
<td>0</td>
<td>1</td>
<td>25Feb2008 15:14</td>
<td>Typical case</td>
<td>12X10</td>
<td>1.75</td>
<td>1.75</td>
</tr>
<tr>
<td>GetRentPerSqFt</td>
<td>0</td>
<td>2</td>
<td>25Feb2008 15:16</td>
<td>Malformed size</td>
<td>bad data</td>
<td>1.75</td>
<td>0</td>
</tr>
<tr>
<td>GetRentPerSqFt</td>
<td>-1</td>
<td>3</td>
<td>25Feb2008 12:03</td>
<td>Negative rent</td>
<td>15 X 5</td>
<td>-2</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 7 – Sample Test Query Results
Another bridge, similar in concept but different in implementation, can be built to test class methods which read and save a Booth object. For instance, the following testing function for checking the basic create, read, update, and delete methods by returning True when the method indicated by CRUDChoice fails:

```vba
Function tstBoothCRUDFailed(BoothID As String, BoothSize As String, RentPerSqFt As Currency, CRUDChoice As String) As Boolean
    Dim tstBooth1 As Booth, tstBooth2 As Booth
    tstBoothCRUDFailed = True
    On Error GoTo tstBoothCrudErr
    Select Case CRUDChoice
        Case Is = "Create"
            'Code hidden to save space'
        Case Is = "Read"
            'Code hidden to save space'
        Case Is = "Update"
            'Code hidden to save space'
        Case Is = "Delete"
            'Code hidden to save space'
        Case Else
            tstBoothCRUDFailed = True
    End Select
    tstBoothCrudExit:
        Exit Function
    tstBoothCrudErr:
        tstBoothCrudFailed = True
        Resume tstBoothCrudExit
    End Function
```

This function is driven by a table of tests and a query definition which calls the testing function with the parameters for each row in the testing suite. A sample run is shown in Figure 9:

<table>
<thead>
<tr>
<th>TestSet</th>
<th>Test Failed</th>
<th>Test PK</th>
<th>TestAdded DT</th>
<th>Test Description</th>
<th>Booth ID</th>
<th>Booth Size</th>
<th>RentPer SqFt</th>
<th>CRUD Choice</th>
<th>Exp DBErr</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoothCRUD</td>
<td>0</td>
<td>1</td>
<td>05Mar2008 10:45</td>
<td>Add a booth</td>
<td>tst1</td>
<td>12X10</td>
<td>3</td>
<td>Create</td>
<td>0</td>
</tr>
<tr>
<td>BoothCRUD</td>
<td>0</td>
<td>2</td>
<td>05Mar2008 10:46</td>
<td>Add same boothID</td>
<td>tst1</td>
<td>4X5</td>
<td>5.5</td>
<td>Create</td>
<td>-1</td>
</tr>
<tr>
<td>BoothCRUD</td>
<td>-1</td>
<td>3</td>
<td>05Mar2008 10:47</td>
<td>Read booth</td>
<td>tst1</td>
<td>12x10</td>
<td>3</td>
<td>Read</td>
<td>0</td>
</tr>
<tr>
<td>BoothCRUD</td>
<td>-1</td>
<td>4</td>
<td>31Mar2008 14:38</td>
<td>Update booth</td>
<td>tst1</td>
<td>12x20</td>
<td>4</td>
<td>Update</td>
<td>0</td>
</tr>
<tr>
<td>BoothCRUD</td>
<td>-1</td>
<td>5</td>
<td>31Mar2008 14:38</td>
<td>Delete booth</td>
<td>tst1</td>
<td>X</td>
<td>0</td>
<td>Delete</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 9 – CRUD Test Results**

Figures 8 and 9 demonstrate some key aspects of TDD:

- The tests ran which means the basic harness is in place; this is progress.
- Most tests failed which is how things should work at first. Now we code to get the tests to pass.
- Unlike previous examples, there is interdependence among tests. The second test is well formed but this CREATE should not work (i.e., get a DB error) in the context of the first test.
- The testing framework and code is beginning to rival the complexity of the code being developed. This does raise some concern that failed tests may be test code issues and not the code being developed, but this is a TDD fact of life. By thinking hard about testing developers get insight into the problem solution.

Despite what might appear to be a complicated set of tables, queries, and helper functions being strewn throughout an application, these components can be organized into groups and hidden from users (or put into a separate database which links to the application database). Forms for selecting separate tests can be developed as can reports which format test results.

As a proof of principle for deploying a test set, consider the following union query which – in a single execution – causes all unit tests to be run and lists only those tests which fail:
SELECT TestSet, TestPK, TestFailed, TestDescription, TestAddedDT
FROM tstqryBoothCRUD
WHERE TestFailed=True;
UNION
SELECT TestSet, TestPK, TestFailed, TestDescription, TestAddedDT
FROM tstqrySetGetRentPerSqFt
WHERE TestFailed=True;

<table>
<thead>
<tr>
<th>TestSet</th>
<th>TestPK</th>
<th>TestFailed</th>
<th>TestDescription</th>
<th>TestAddedDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoothCRUD</td>
<td>3</td>
<td>-1</td>
<td>Read booth</td>
<td>3/5/2008 10:47:00 AM</td>
</tr>
<tr>
<td>BoothCRUD</td>
<td>4</td>
<td>-1</td>
<td>Update booth</td>
<td>3/31/2008 2:38:16 PM</td>
</tr>
<tr>
<td>BoothCRUD</td>
<td>5</td>
<td>-1</td>
<td>Delete booth</td>
<td>3/31/2008 2:38:57 PM</td>
</tr>
<tr>
<td>SetGetREntPerSqFt</td>
<td>3</td>
<td>-1</td>
<td>Negative rent</td>
<td>2/25/2008 12:03:00 PM</td>
</tr>
</tbody>
</table>

Figure 10 – Summarized Test Results

Executing this query provides a one stop check for current status and what needs to be developed. Other reports or inquiries can provide a history of test development, bug identification, and function completions. It takes some up front work, but TDD does pay off at both developer and manager levels.

CONCLUSION

Test driven development is a mind set, not a tool set. Tool sets help a great deal, however, and for enterprise-wide, team oriented projects a strong set of tools is a near necessity and repays the steep learning curve. Within the enterprise or in smaller settings based upon VBA and Access, however, the tools already in place can be used to develop a strong testing suite. By taking the perspective that test components (tables, queries, forms, reports) are as valuable a set of deliverables as solution components, testing moves from an afterthought to a development driver with built-in documentation, replication, and reuse.

We have focused on unit tests for subroutines, class properties, and class methods which require small amounts of bridge or façade testing code to be developed. This code is typically not as complex as the code being tested, but it does require knowledge of the class interfaces and some clever use of Access’s built in queries. Natural areas for extension are automating testing of forms and validation, reports, and queries, especially action queries which change the data. Creating and executing these sorts of test suites without development of significant testing code at least as complex as the components being tested would be a breakthrough.

REFERENCE LIST