Paper 42-27

(Web) Software Development: Best Practices for Developing Enterprise Applications

Greg Barnes Nelson, ThotWave Technologies, Cary, NC
Danny Grasse, ThotWave Technologies, Cary, NC

INTRODUCTION

Software development has a rich history that stems from forty plus years of tried and true practices. From structured programming models to object-oriented methods, the student is often left to her own devices to implement code based on a highly creative – and subjective – design process. In this paper, we will explore some of the foundations of our software development methodologies and apply these to web programming for SAS® applications. We will present coding techniques, lessons learned from our experiences as architects and experiences proven to be too painful to be ignored. Examples will include reference to HTML, JavaScript, SAS/IntrNet®, JavaServer Pages, XML and ODS.

SOFTWARE EXCELLENCE

Despite the fact that software development is inherently a human endeavor – wrought with highly subjective, highly personalized notions – we have, as an industry, attempted to measure the successful programmer through objective measures. Many of these were born out of government standards in the 1960s ((1990; 1992)). Since then, we have seen organizations create metrics with which to measure organizations (e.g., Software Engineering Institute) and people ((DeMarco 1987)). A relatively recent trend in both software engineering and project management circles has been the increased reliance on generic methodologies from both non-profit organizations (e.g., Project Management Institute) and for-profit companies specializing in the development of "standards" (e.g., The Gartner Group).

Methodologies

Many of these methodologies, practices and "process maps" focus on the way in which we "do" software development – with only the most general guidance on "how" we might approach the problems for the practicing programmer. Despite the proliferation of software for everyday applications – from on-line ordering to medical devices – many of these approaches rely heavily on "processes". For example, in Figure 1 we see the traditional software engineering flow diagram, which doesn't prove highly useful when asked how we might approach the design and development of the actual code.

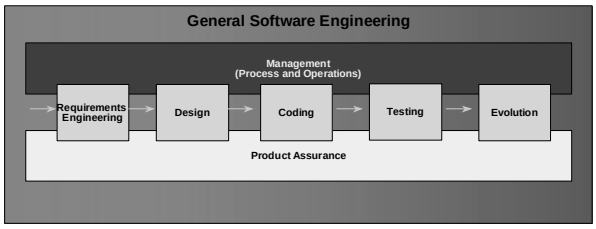


Figure 1. Software Engineering Process/ Life Cycle

Similarly, we see traces of these "process" approaches in decision support applications. Figure 2 shows us a typical "methodology" for how we might build a data warehouse. Here, the focus is on making sure that the component parts of the warehouse all have their place and that there are tools and techniques appropriate to each specific "component" or box.

A prime example of this approach is one where the methodology traces to specific processes that are implemented – complete with tools and technologies (templates) to support the methodology.

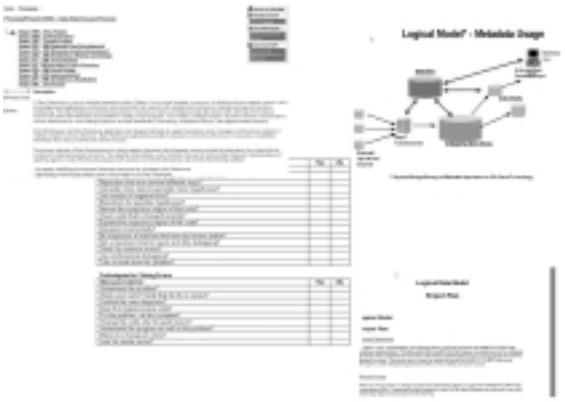


Figure 2. Formal Methodologies Combine Approaches with Tools and Techniques

Principles, Techniques and Practices

From the discussion above, it seems safe to say that some of the methodologies – whether it is an approach to development like Rational's Unified Process or a specific methodology (as in the data warehousing methodology defined above), we are required as a developer to know where we are in the process to determine which process, tool or technique is appropriate. These approaches serve an important role in software development, but I would argue that most of these serve the people who manage the projects and are less important for the people who actually deliver the software – leaving a relatively large gap between what managers need and want for organizing a project and the developers who need specific and tactical best practices for how they might do their jobs.

It is important to note at this point the distinction that exists between what might be considered a general "principle" of software development and a specific technique appropriate to a language or tool. We make no attempt to document the specific techniques in SAS for building reliable, robust software – but rather cite examples, which support the general principles that seem to be appropriate here.

In this paper, we do not intend to focus on a specific methodology, phase, process, type of software project, nor do we care about the kind of technology or architecture that is being discussed – but rather, we focus on concepts that extend beyond the phase or specific architecture of the software project – that is, the principles which guide our behavior as developers. Our belief is that if we focus on specific phases, the developer, if asked to work on an unfamiliar phase, would be lost without the process map to guide her. This is a common sense heuristic that not only makes sense, but also is highly appropriate to web development – where sometimes decisions in architecture and the interrelatedness of technologies make it challenging for developers.

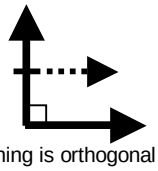
7 PRACTICES FOR EXCELLENT SOFTWARE

Specifically, we will discuss seven practices that transcend phase, architecture and type of project. These include:

- Orthogonal
- DRY
- Model-View
- Meta-programming
- Automatic
- Test First
- Refactoring

ORTHOGONAL

Orthogonal is a term meaning "at 90-degrees" – or perpendicular. In geometry, two lines are orthogonal if they meet at 90 degrees. As we can see from the Figure on the right, changes in one axis, have no effect on the other axis.



In software development we might say that something is orthogonal if you make a change in one part of the system, the change should have no effect on other parts of the system. For example, we might have an application that is built on a specific database (say Oracle) – when a change in the underlying database doesn't have an effect on the user interface – that is said to be orthogonal.

Non-orthogonal systems are inherently difficult to change and control. Seemingly simple fixes have far reaching effects in these systems and are difficult to test since the impact seems pervasive.

Larry Constantine and Edward Yourdon ((Constantine 1979)) first introduced the concept of orthogonality in the 1970s in the context of software development. In this seminal work, they discussed the concepts of cohesion and coupling as one of the best ways of measuring the inherent maintainability and adaptability of a software system. Coupling is a measure of how interrelated two software components are. Cohesion is a measure of how related the functions performed by a software component are. We want low coupling and high cohesion. High coupling (or 'tight' coupling) implies that, when we change a component, changes to other components are likely. Low cohesion implies difficulty in isolating the causes of errors or places to adapt to meet new requirements.

Techniques

There are several ways that we as developers can maintain orthogonality. In general, we do this by keeping our code decoupled. That is, we write code that has specialized functions. For SAS programmers, we can do this by writing macros or methods that play special roles in our code. For example a single macro may be the only way in which we update a global variable. By hiding the interfaces – or keeping the code isolated from other components – we ensure that changes in the way the global variable is instantiated, changed or destroyed is isolated to this particular macro. It is of particular importance here to avoid similar functions (macros). In fact, our goal often is to generalize the macros so that we can avoid duplication of code (see DRY below).

In SCL applications, we can also keep our code decoupled by avoiding global environment variables. It is all too easy to allow global variables to be updated by multiple programs. Avoiding them altogether helps maintain orthogonal code.

The goal of this principle is simple: Eliminate effects of unrelated things. By creating code that is independent and self-contained, we create an environment that is far easier to test and perhaps more importantly, to implement changes without introducing defects in our codebase.

Examples

In n-tier applications such as those which have web-based presentation layers, it is easy to spot non-orthogonal components. For example, many code walkthroughs unearth ASP (ActiveServer Pages) or JSP (JavaServer Pages) with complex business logic or even database access mixed in the HTML (presentation layer). Later, we will also discuss this concept in the context of Model-View architecture, however it also demonstrates the idea of orthogonality, as a simple change in the table structure in our database, would yield highly unstable interfaces to our user population.

Another subtler example might be in how we architect our applications – especially in enterprise applications where system boundaries expose volatilities in the way that we set up our directories, libraries and external files. Since SAS is one of the most portable development environments (code can be ported to over 22 operating systems), we need to make sure that any change in our architecture doesn't affect the application. We can do this by planning for portability and scalability in the way that we set our directory structures up and how we call our autoexec.sas files. We most often do this through the use of environment variables that define the host, macro variables that define the high level qualifier of the directory. Even using simple, reusable macros that define the hostname can make it easy to split off application logic and database access across machines.

Finally, for a good example of keeping our code independent from module to module in the Java world, we look to "properties" files. Here, code is kept that is usually run-time parameters for the application (see the section on Meta-programming). "Properties" files allow code that "drives" applications to be split up so that developers can work on multiple components without repeating each other, yet keep the code consistent.

Importance

In general, we want to maintain orthogonality in our applications so that is it easy to change things, we can reuse code (specific, well defined responsibilities) and ensure defects are not introduced because we haven't done our job at protecting the interfaces between components in an application. By doing this we can see marked improvements in efficiency / productivity (localized development and testing). In addition to being a good development practice, we also tend to reduce our risks as problems are isolated; they are less susceptible to defects, we achieve better results (easier to test) and they tend to be more portable (between languages, vendors, products, etc.)

DRY (DON'T REPEAT YOURSELF)

Software development's "holy grail" is reusability. Often times, we fall short of this goal because of changes in technology, shifting opinion about approaches and even our own maturity as developers lead us to improve – and drastically change – our code. One of the dangers of this "continual improvement" is that in an effort to do things better, we often find ourselves duplicating information. The principle of DRY – or Don't Repeat Yourself – is seen in code for a variety of reasons. As Hunt and Thomas describe ((Hunt)), developers tend to repeat themselves for a variety of reasons – legitimate or otherwise:

Imposed duplication. Developers feel like they have no choice. Because of process issues, imposed standards, multiple platforms, or languages seem to require "copies" of code.

Inadvertent duplication. Developers don't realize that they are duplicating information. Often times this is caused by incomplete information, lack of code walkthroughs or even poor design on the part of developers.

Impatient duplication. Sometimes it is just easier to duplicate code, rather than refactoring or waiting on other developers to coordinate the shared information.

Interdeveloper duplication. Perhaps the most common type of duplication occurs when multiple developers are working on the same system and little attention is paid to managing the code library.

Techniques

As discussed above, the better we isolate the dependencies among code, the chances are that we can and should be able to use these generalized routines again. In SAS we have SCL methods and Base macros; in Java, we have methods (JSP Scriptlets and custom tags) and classes that promote reuse.

There are several non-technical approaches that may also prove useful when any of the reasons that we cited above characterize our code. For example, a simple, but effective way to track duplicate code - whether imposed or impatient) is simply to maintain a developer's notebook that logs known developer "issues". This surfaces the duplication early (while the duplication is occurring) and is more likely to be address when refactoring occurs since it has been noted.

In addition, two other techniques make it easier to catch duplication due to inter-developer or inadvertent duplication. These include appointing a librarian that is responsible to cataloguing all code (from a functional perspective) as well as diligent code walk-through sessions ((Yourdon 1979)). Each serves an important reminder for the developer through the development cycle to be mindful of the fact that software development is a highly collaborative effort, not one merely a result of acts of heroism. Finally, it should be noted that if duplication is a result of "standards", these standards should be reviewed for reasonableness.

Examples

In SAS, it is easy to come up with clear examples of how we can prevent duplication in our systems with good design principles. For example, pretty much every SAS programmer I know has developed or at least utilized small utility programs or macros that do very specialized things. Checking the number of observations before a critical step or for the existence of a macro variable before one changes it, are things that we do often enough to require good code libraries. Often overlooked, these cataloging efforts can trim several hours, if not days off of a project.

Another example that is often overlooked is the calculation of a business rule. In a large system – especially with data warehousing applications that segment responsibility from extract, transformation, load and somewhere down the line, exploitation, we may need to utilize a calculation. Deciding where this occurs and how the code is implemented in different languages (PL/SQL versus SAS macros versus Java) can introduce similar functions across step boundaries. There is not much the programmer can do to avoid ALL of these, however, where we implement a business rule/ calculation in the same language (e.g., SAS macros), we can take advantage of recent improvements such as Stored Processes and the Java APIs now available to us.

Importance

Although the importance of DRY is probably clear to the reader by this point, it is worth noting that if we make it easy for code to be reused, it is more likely that it will be. If we can reuse code – we are reusing knowledge and it will be much easier to test, to maintain and to extend.

MODEL-VIEW (MVC)

In an earlier paper ([Barnes Nelson, 1999 #46]), Barnes Nelson discusses how SAS applications can take advantage of the Model-View approach. We reference that here, as some of this content has been adapted from that paper.

The concept of Model-Viewer-Controller (MVC) originated in the late 1970's at Xerox PARC to support the Smalltalk-80 object-oriented programming interface ((Simon 1995)). It is used as a framework to help us understand and implement graphical user interfaces (GUI) and has been reused to varying degrees in other programming languages ((Gamma 1995)).

Model-View-Controller (MVC) or, as it is often referred to -- Model-View, is a way of developing applications whose primary purpose is to provide a clear separation between the presentation of the GUI, the application logic and data. This is the essence of the MVC paradigm.

As we get closer to realizing this separation in our applications, we get closer to the pure goal in Object-Oriented Programming: reusability. By building applications that are independent of the data, we can attach our model to new data without rewriting the code. By assigning responsibility to the Model, the Viewer and the Controller we take the tasks of modeling the external world, the visual feedback and user input respectively. In most discussions of MVC, the Viewer and the Controller are tightly coupled and often the latter gets dropped in favor of the Model-View perspective instead. Regardless, we see the clear separation of the areas of responsibilities for the application framework.

In a nutshell, Model-View allows us to develop or focus on independent, well-organized modules. We essentially distinguish between how something is displayed (presentation), from where it came from (database), and the business rules applied (application logic) to the data prior to presentation.

Techniques

The way that we think about the separation of the three layers in the application helps ensure that we adhere to this approach to software development. The web environment gives us a readily understandable example.

Mixing Logic and Presentation is Easy (too easy!)

In JavaServer Pages (JSP), ActiveServer Pages (ASP) or even SAS/IntrNet, we have to pay specific attention to making sure that we don't violate this principle. Often the business logic is tied with how we present content. For example, the following JSP code absolutely violates this rule by tying what rows are returned from the dataset along side the code that writes the HTML.

```
<%!-- START BAD EXAMPLE  --%>
<table>
<%
  int patientColIndex =
dataSetInfoInterface.getVariableIndex("patient");
  int dateColIndex =
dataSetInfoInterface.getVariableIndex("date");
  int BPColIndex =
dataSetInfoInterface.getVariableIndex("BP");


  String currentPatient = "";
  int numRows = dataSetInterface.countRows (0);
  for (int i = 1; i <= numRows; i++) //iterate
through the rows
  {
    String patient =
((String)dataSetInterface.getCell (i,
patientColIndex)).trim ();
    if (!patient.equals (currentPatient))
    {
      currentPatient = patient;
%>
  <tr>
    <td><%=patient %></td>
    <td colspan='2'> </td>
  </tr>
```

```
<%
    }
    String date = dataSetInterface.getFormattedCell
(i, dateColIndex).trim();
    String BP = ((String)dataSetInterface.getCell (i,
BPColIndex)).trim ();
%>
  <tr>
    <td> </td>
    <td><%=date %></td>
    <td><%=BP %></td>
  </tr>
<%
  }
%>
</table>
<%!-- END BAD EXAMPLE  --%>
```

A cleaner approach is the following code, where we simply call the method that retrieves the rows when we need it, but has an independent coupling with the presentation. We can say that something is loosely coupled or independent when published or approved "interfaces" protect code from leaking into one another.

```
<%!-- START BETTER EXAMPLE  --%>
<table>
<%
  Iterator patientList =
dataRetrieverObject.getPatients ();
  while (patientList.hasNext ())
  {
    Patient patient = (Patient)patientList.next ();
%>
  <tr>
    <td><%=patient.getName () %></td>
    <td colspan='2'> </td>
  </tr>
<%
    Iterator dataList = patient.getDataList ();
    while (dataList.hasNext ())
    {
        // using a Map here allows for data to be
added
        //  to Patient object by dataRetrieverObject
        //  AND extracted by jsp code
        //  without changing the Patient class API
      Map patientData = (Map)dataList.next ();
%>
  <tr>
    <td> </td>
    <td><%=patientData.get ("data") %></td>
    <td><%=patientData.get ("BP") %></td>
  </tr>
<%
    }
  }
%>
</table>
<%!-- END BETTER EXAMPLE  --%>
```

Where Should Formatted Variables be Formatted?

A second, but less confident example includes where and how we format data. Specifically, let's say that we are required to format a number as Dollar12.2 (dollar signs, commas and 2 numbers to the right of the decimal sign). It isn't always clear where the formatting of that variable should occur. For example, we could change the format in the SAS dataset and have that flow through to all reporting and end-user applications. That way, you don't have to change it in every application and global changes are easy.

On the other hand, formatting of the variable when it's being presented makes good, logical sense. Formatting is a presentation task anyway, right?

Like most things in SAS, there is multiple ways to approach any one problem. Let's look at the possible scenarios and their pros and cons.

Location of Logic	Benefits	Drawbacks
Database layer (SAS dataset)	❖ Use the metadata in the data to drive applications. ❖ Changes in downstream applications (e.g., reporting) can be made globally. ❖ Multiple consumers of data can see the data the same way. ❖ The client can always apply another format.	❖ Databases may change and the resulting metadata may not have the same features. ❖ Database changes may be too pervasive. ❖ There may not be a global standard of how something should be stored (and subsequently presented).
Business Logic (SAS Macro or Java servlet)	❖ Business logic about how a variable is calculated is close to how it is sent to the client (easier for a single programmer to handle). ❖ Impact on database changes is reduced.	❖ May not be obvious that the logic about what is being presented is not in the "view". ❖ Changes in the viewer may override its effects (DRY).
Presentation Layer (JSP code)	❖ Changes in how a variable "performs" in multiple reports can be localized. ❖ Have complete control over how something will look as it is presented. ❖ Reduce dependency on database.	❖ Global changes are difficult to manage since each "report" has to be touched. ❖ Data driven applications are easier to maintain. ❖ Ties presentation interface too closely with specific "variables".

But as our dear friend Jack says, "there is always a right thing to do". You just have to understand the tradeoffs at play and what the relative importance of the benefits and the drawbacks. From a purist's perspective, I guess we would have to argue that any presentation characteristic should be managed at the viewer. Good thing we aren't always that myopic.

XML for the Web – Model-View at its Best

The final example we show will make you marvel (ok, it worked for us.) Perhaps one of the cleanest examples of Model-View that we have seen involves using XML to deliver data and XSL to present the

data. In a related paper ((Barnes Nelson 2000)), we introduced XML in the context of SAS applications. Here, we showed an example of how you can take a data, process it with SAS (macro) and deliver the XML to the client. From there, XSL takes over and formats it as an HTML table that can be sorted on the client.

We've seen no better example of the separation of the model from the viewer than this. If, for example, we wanted to change the way it looked, we need only change the XSL. If we wanted to restrict the data, make a change to the calculations or change the content of the table, we do that in the model.

Importance

Clearly, this approach gives us some specific, tangible benefits. These include faster development, higher quality output, easier maintenance, reduced costs, increased scalability, better information structures and increased adaptability. Some of these benefits are difficult to measure, with that said, in our simple examples we have shown the following benefits.

> **View independence:** multiple presentations can be built (HTML, WML, PDF) as different views on the same underlying business logic and data structures.
>
> **Model independence:** Allows developers to change and evolve data structures or file formats without changing how the data is displayed or processed in the rest of the program, as well as introduce persistence, remote databases, and sharing.
>
> **Reusable logic:** Once implemented, different presenters can reuse these in multiple applications (e.g., XML can be reused for wireless devices).

Perhaps most important is that from a personnel perspective, we can reposition our current SAS programming staff to take advantage of this new technology without having to relearn WML, Java or .NET. We allow the development of new viewers on top of our existing investment in these applications.

META-PROGRAMMING

Out of all of the languages that we have seen, SAS seems to support the concept of meta-programming best. Of course, those with a deeper appreciation of non-SAS languages would argue fiercely, but I would posit that SAS has built the languages with that concept in mind – and expect developers to use it.

Meta-programming is a principle which calls for removing all of the details in the code and moving that to something the code can look for when it needs it. Some people do this through data-driven applications (such as a lookup table); others do this through exploiting the SQL dictionary tables to query known information about tables; and good programmers do this with symbols (macros) and formats.

Metadata-driven applications look for the code to be generalized and the specific details to be something we supply just in time. A good example is in the simple IF-THEN statement. Instead of providing the conditions and the action in the code, we can look for the conditions in the data – along with what to do with them.

Another example in web applications that rely on SAS for compute services is the "autoexec" file. We learned a long time ago that you should never place "hard-coded" values in your code that make it non-portable or non-extensible. If we isolate the dependencies (host specific information like paths or things that can be gleaned from the environment itself), then we stand a better chance to reuse and extend the application when the time comes.

In the Java world, we often see this principle expressed in the form of a properties file or some XML-based configuration file. These contain configuration or run-time options that would otherwise be changed in the code (and require a re-compile).

As Hunt and Thomas suggest – "put abstractions in the code, details in the metadata" – which helps us plan for change. ((Hunt)).

Importance

It may seem obvious that the reason we do this is so that we can anticipate change – especially when it comes to data – which we all know does often. Requirements change and it's often easier to make change happen at the interfaces (metadata) rather than having to touch the code. We often say of our applications when asked if the data can be displayed "it's just a matter of getting the right data, the code's already in place."

AUTOMATIC

"Some of the laziest programmers write the best code!"

You've probably heard that sentiment expressed as it pretty accurately relays the concept of writing code so that it does all of the hard work for you. We remember when we had to write a series of PROC PRINTs for every dataset in a library. Having to copy and paste all of the code and changing the MEMBER name for each dataset got tedious. Ah, the beauty of macros!

But the principle of automation goes well beyond satisfying the needs of the lazy. It has to do with making sure that someone can accurately and reliably reproduce the task at hand. Running a job that updates the data warehouse exactly a 2:00 a.m. GMT is much easier to achieve through automation rather than setting the alarm clock.

This principle extends through just about every aspect of software development that we can think of: setting up the workstations that we use to develop on, creating the server libraries, automating the backups, documenting our code and even building our test scripts. The challenge with automation is knowing when it's worth the trouble. For example, writing code that takes days to write to automate a task that is seldom used would be an exercise in egoism. Similarly, things that we do every day for years on end that no one ever seems to pay much attention to make for likely candidates – especially when they can reduce human error.

Build a Macro that Calls a Macro

We once wrote a macro that took a dataset and a macro name as its parameters. Its job was to loop through the variables and the observations and build separate calls to another macro. The variables in the dataset were simply the parameters that were used in the macro and the observations represented the number of unique ways that we wanted to call the macro. This may seem overly complicated – why not call the macro in the code itself? The reason for us is that we didn't know how many times we needed to call it – it was data driven (in our case, members in a library). Little did we know then that we would be using that tool now in just about every application we build – 12 years later!

Java docs and other shell tricks

Documentation seems to be one of the very last things we do as developers. Relegated to the final hour, it often goes undone and short of the mark (i.e., creating shared understanding of the function and form). Java has one of the best models for documentation – build it as you go. By commenting code in specific ways, the JAVADOC tool allows you to create HTML based documentation, complete with hyperlinks, dependencies among classes, fully documented methods, etc.

Of course, one could argue that in SAS there are too many different language elements to support (major PROCs, DATA Step) and even different underlying languages (Base, Macro, SQL, SCL). Clearly you have to understand where the biggest bang for your buck is going to lie. For SCL based applications, Qualex Consulting has provided us with a tool to help document our code and datasets. But for Macros, SQL and other "base" language elements, we are left to achieve our own standards in documentation.

For a poor-man's version of the JAVADOC tool that is relevant to SAS applications we look to good text processing tools like awk and sed (UNIX geek tools). We developed a program using awk several

years ago to parse a text file (.sas file) and generate documentation based on pattern recognition. Here we simply looked for the most typical in-line commenting patterns:

```
*…;     single line comments
/* … */ block comments
%* ..;  for macro style comments
```

Based on these patterns, awk would parse the files and generate text files ready for inclusion in our developer's notes.

Of course, automation in web development can be seen in other places – these include code generation (macros); HTML generation (SAS/IntrNet, SCL, Macros, JSP, ASP, XSL, etc.); configuration management (generation scripts); and as we will explore in the next section: automatic testing.

TEST-FIRST

If documentation is relegated to the last possible moment, testing often never occurs – or occurs only in the hands of users. We posit here that if you make it easy for people to test, they will be more likely to actually do it. One method that we learned from the XP world (eXtreme Programming) is the concept of Test—First. In their view, you write a test case to prove the code does what it is supposed to do. For example, if we write a simple macro that reads a dataset, calculates a new variable and then writes the new variable out to memory as an array of macro variables (one for each observation in the dataset), we would build the test case iteratively:

Step 1. Confirm the dataset is valid (exists and has >0 observations; constraints on variable types, libraries, etc.)

Step 2. Read the dataset and confirm the required variables for the calculation have valid constraints (non-missing, all signed appropriately, valid ranges, intra- and inter-variable dependencies)

Step 3. Create the variable of interest and validate the expected value (non-zero, non-missing, range, etc.)

Step 4. Create the resultant macro array and validate

The number of observations in the dataset equals the number of newly create macro variables (dictionary tables can verify this)

Create a temporary dataset to hold the values from the data step and do a compare against the SQL output (SQL gives us an easy way to write an array of macro values)

Create a macro symbol that contains the number of macros we should find and compare against the number of observations in the dataset.

All of these steps are fairly trivial in and of themselves. But seldom have we seen someone actually think through the process of what should be tested (except for regulated applications like FDA submissions in Pharmaceutical applications). If followed, this process not only leads to better software and better requirements definition, but also should save considerable time and effort as we don't write any code until the test cases have been written. We would also argue that having to write these test cases once, you would soon establish very good libraries or reusable tests (we hate to write code twice if we ever think we will need to write the code again).

Moreover, if the tests are written first, then any change that occurs with the code can automatically be checked against the requirements because the requirements drive the test cases. It is full-circle development. After all, if you make it easy for developers to test – they will.

Techniques

Here, we can summarize the Test-First principle in the following way:

Test first. Test often. Test automatically. Write tests to prove the requirement, then the code. Now that the test is written, include this as part of the code (Debug=Y) so that it gets run every time a code change is made.

Test against your contract. Macros and methods both have the advantage of a "published and approved" interface. These interfaces serve as the gateway to the code. Validate the interfaces and your code will serve you a long time.

Design to Test. By building your code in such a way that makes it easy to test, you make it easy for the development staff to actually do the testing.

Unit test in the code. Similarly, creating code that allows return codes in the logs and applications specific ERROR or WARNING indicators in the LOG, the developer let's every one know what the code is doing.

Functional / User Acceptance. Because we write our code based on a test case and test cases are written against requirements, we should have code contracts in place with the end users – or those that create requirements. Functional and user acceptance testing is a necessary, but not sufficient requirement for good code.

Stress/ Load Test. Finally, although it's not always known at the time the code is written, you should always stress test your code against unrealistically high volumes of data and/or users. This tells the user community what the valid constraints on your application might be – before your code breaks. Recently, our experience with this saved us a very potentially embarrassing situation with a client. By understanding what the load patterns were on a web application, we were able to refactor the connections to SAS (from Java) to improve performance.

Importance

Although we are not sure where the saying came from, the following certainly seems true of our experience:

It costs a lot less to fix a defect or bug in requirements or design that it does if the users find it. (Not to mention the political side-effects!)

Location of Fix	Cost
Requirements definition	$1.00
Technical Design	$10.00
Coding (Implementation)	$100.00
Integration or System Test	$1,000.00
End User (Production)	$10,000.00

Remember, if you will, the story of Babe Ruth pointing to the right field wall in the 1932 World Series and "calling his shot". What comes to mind for us is that fact that Babe knew where he was going. Like software requirements, the test-first principle helps us see where we need to go. After all, if you know where you want to go, you stand a better chance of getting there.

REFACTORING

Despite your best efforts at writing good code, making sure that the requirements were met, automating those tasks around you which potentially introduced errors or bugs, we often have to go back to the drawing board and rewrite, rework and re-architect the code. This healthy endeavor, which leads to better practices and better software, is known as "refactoring". Writing software is an organic process – replete with our propensity for error. However, there are a lot of reasons why we might go back to the drawing board and take a second or third look at our code. As Hunt and Thomas describe ((Hunt)), there are a number of things that force us to refactor:

Duplication. We've inadvertently, or otherwise, introduced duplication in our system.

Non-orthogonal design. We have implemented code that is

too pervasive making it difficult to change.

Outdated knowledge. We simply know more that we did when we started.

Performance. Often we find that our code just doesn't perform as it should.

Refactoring is the process of revisiting earlier decisions – whether they be architectural, design, relationships between classes or whatever – and retooling the code so that it performs better, is easier to maintain and defect free. Except for the performance issue mentioned above, no one is likely to notice the fruits of your refactoring efforts except other developers. A cautionary tale here is that make sure that the effort to refactor makes sense. Having been a developer for many years and managing even longer, we find ourselves taking the code away – or in my own case, having it ripped away – as we plead "but it's just not ready yet, we have one more thing to do". Since developing software is an inherently organic process sometimes we just have to say that we have done our best with the information we had at the time.

Techniques

The best tips on how to refactor come to us from Martin Fowlers seminal work on Refactoring. Here, he describes three very simple tips:

1. *Don't try to refactor and add functionality at the same time*. It may seem like a good idea to wait until some piece of new functionality to revisit that code you've wanted to touch, but try and resist. Make a special effort to fix it without having too many moving parts at once. One practical piece of advise here is to make sure that the planned functionality doesn't change the underpinnings of your code so dramatically so as to waste time or money.
2. Make sure that you have good tests before you begin to refactor. As mentioned above, if we follow the test-first principle, we should be ahead of the game here. By having good tests that produce reliable results, you'll know soon enough if your refactoring has introduced defects.
3. Take short, deliberate steps. Here Fowler describes examples primarily from object-oriented languages (C++ and Java), but the extension is SAS is obvious: don't try to change everything at once! If you want to move something from an IF-THEN block to a dataset, read the dataset, validate the variables,and get the unique values before ever trying to take out the IF-THEN statements. Slow, deliberate changes will be easier to back out if you have a problem.

Examples

In web programming, we have come across several examples that we think are good lessons for us. The first really has to do with extending a piece of code for something that was not intended and the second illustrates how we interpret changes to a system so that we can anticipate future changes.

Client versus Server Processing

Something that is not always obvious in any application is where something should take place – the client or the server. An example that comes to mind is being able to sort an HTML table on the client (Browser) without repeat visits back to the server. After heated discussions about who should be doing the work, we thought the tradeoffs of scalability and ease of use (for the end user) outweighed the potential downsides (complexity for the developer).

In this application, we wanted to be able to sort the data on the client by having the end user click on a column heading. By allowing for the sorting to occur on the client, the response time would be much faster and the user could "play" with their data in a more real-time fashion (than is typically doable on the web). Having successfully implemented this code for said application, we thought we understood the tradeoffs and by using XML and XSL, we achieved what we intended and the client was happy.

Soon thereafter, we were designing a similar application and decided to use the same approach. Once we loaded real data into the application, we found a dismal discovery: it was slow! The difference: Application 1 always had less than 500 rows in the table; Application 2 usually had more than 2000 rows. The discovery forced us to refactor. Since XSL as a technology was terribly slow, we had to rethink how we were going to solve the problem. After all – the expectation had been set with the end users, they would get client-side sorting. The refactoring effort took us down several paths including more XML, HTML with JavaScript events and even the dreadful realization that we might have to go back to server-side sorts. Fortunately, one of the authors developed an object-oriented solution that performed beautifully. Even with 10,000 rows, we could sort in less than 3 seconds – which was our performance benchmark.

Designing for growth (or unintended uses)

In the above example, the code was completed and then we tried to use it in a second application. There, it wasn't so much of a problem that we wanted to use it for something that was unintended, but we didn't know the limits of the code. Our second example takes us to a common data set design problem – do we make our data fat and short (lots of variables – one for each unit of measure), or do we make it long and skinny (turn our variables of interest into one variable and have a class variable that defines the observation). A common example follows.

Suppose we have a dataset that contains the following variables:

Date	Date of Visit	MMDDYY6.
Patient	Patient ID	Char4.
BP	Blood Pressure	8.

Our application uses the information from the variables to populate a list box on a HTML form (a list box for the analytic variables only).

Then along comes a request to add heart rate to the table. Normally, adding a column to the table would satisfy these requests. Often, it is important to think about the implications of the request – "if I simply add a column, then I have to go into the HTML and add some code around getting the Heart Rate into a list box as well. The code is tightly coupled with the data.

Instead, we would argue that this request should be analyzed for potential refactoring. If we were to turn the dataset on its side and have the unit of observation become Date, Patient and Variable Type (versus Date and Patient), then we can automatically build the list boxes directly from the Variable Type column.

We are not suggesting that this is the appropriate design decision for all applications – that would be categorically wrong. In this example, the benefit of having the unit of observation be Date and Patient is that if I wanted to know anything about the patient, I would simply have the look at one row per date, which would make processing easier for reporting. Knowing which design is appropriate and the tradeoffs become crucial to a developer's skill set.

Importance

Refactoring helps us look at code through a critical eye. We revisit previous decisions so that we can improve the code for performance reasons as well as readability and to make it easier to test. Refactoring isn't a stage in the project plan; it is part of the development life cycle. Once again, one of our favorite tools in our arsenal is simply the code-walkthrough sessions and developer

notes. By documenting as we go, known deficiencies and peer comments, we make better software. Just remember what it says on the shampoo bottle: lather, rinse, and repeat.

CONCLUSION

The following statement, lifted from an email, sums up this paper better than we can:

The four R's as defined by Jack.

> ***Reasonable -*** *Is the application, program, or module reasonable? That is, does it produce results that are reasonable in the context in which it is used?*
>
> ***Reproducible*** *- Are the results reproducible? That is, when the application, program, or module is run multiple times with the same input, will it result in the same results?*
>
> ***Reliable -*** *Is the application, program, or module reliable? That is, can a user of the object expect it to be available and work as advertised?*
>
> ***Robust -*** *Is the application, program, or module robust? That is, will the object continue to work under conditions beyond the original specifications?*

Things that satisfy all four of these R's are truly world class.

ACKNOWLEDGMENTS

The author would like to sincerely thank several people for their guidance and thoughtful review of this manuscript. Specifically, we would like to thank Jack Shoemaker, Jeff Wright, John Leveille for showing us how to build world-class software and to Dave Hamilton for his gentle review of this paper. In addition, two of the most influential sources for inspiration have been my long-time friend and mentor – Ian Whitlock as well as the published works of Andy Hunt, Dave Thomas and Fred Brooks.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Please feel free to contact the author at:

Greg Barnes Nelson
greg@thotwave.com
117 Edinburgh South
Suite 202
Cary, NC 27511
919.465.0322 - Phone
919.465.0323 - Fax

BIBLIOGRAPHY

Barnes Nelson, G. (2000). XML and SAS: An Advanced Tutorial. SAS Users Group International, Indianapolis, IN, SAS Institute.

Constantine, L. a. Y., E. (1979). Structured Design. Englewood Cliffs, N.J., Prentice Hall.

DeMarco, T. a. L., Timothy (1987). Peopleware: Productive Projects and Teams. New York, Dorset House.

Gamma, E., et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA, Addison-Wesley.

Hunt, A. a. T., David The Pragmatic Programmer.

NASA (1990). Manager's Handbook for Software Development, Revision 1. Greenbelt, Maryland, Goddard Space Flight Center, NASA.

NASA (1992). Recommended Approach to Software Development, Revision 3. Greenbelt, Maryland, NASA Goddard Space Flight Center, NASA.

Simon, L. (1995). The Art and Science of Smalltalk: An Introduction to Object-Oriented Programming Using VisualWorks. London, UK, Prentice-Hall.

Yourdon, E. (1979). Structured walkthroughs. Englewood Cliffs, N.J., Prentice-Hall.